



Vũ Hữu Tiệp

Machine Learning cơ bản

machinelearningcoban.com

Vũ Hữu Tiệp

Machine Learning cơ bản

Order ebook tại <https://machinelearningcoban.com/ebook/>

Blog: <https://machinelearningcoban.com>

Facebook Page: <https://www.facebook.com/machinelearningbasicvn/>

Facebook Group: <https://www.facebook.com/groups/machinelearningcoban/>

Interactive Learning: <https://fundaml.com>

Last update:

March 27, 2018

Lời tác giả

Những năm gần đây, *trí tuệ nhân tạo* (*artificial intelligence–AI*) nổi lên như một bằng chứng của cuộc cách mạng công nghiệp lần thứ tư (1–động cơ hơi nước, 2–năng lượng điện, 3–công nghệ thông tin). Trí tuệ nhân tạo đã và đang trở thành thành phần cốt lõi trong các hệ thống công nghệ cao. Nó đã len lỏi vào hầu hết các lĩnh vực trong đời sống mà có thể chúng ta không nhận ra. Xe tự hành của Google và Tesla, hệ thống tự tag khuôn mặt trong ảnh của Facebook; trợ lý ảo Siri của Apple, hệ thống gợi ý sản phẩm của Amazon, hệ thống gợi ý phim của Netflix, hệ thống dịch đa ngôn ngữ Google Translate, máy chơi cờ vây AlphaGo và gần đây là AlphaGo Zero của Google DeepMind, v.v., chỉ là một vài ứng dụng nổi bật trong vô vàn những ứng dụng của trí tuệ nhân tạo.

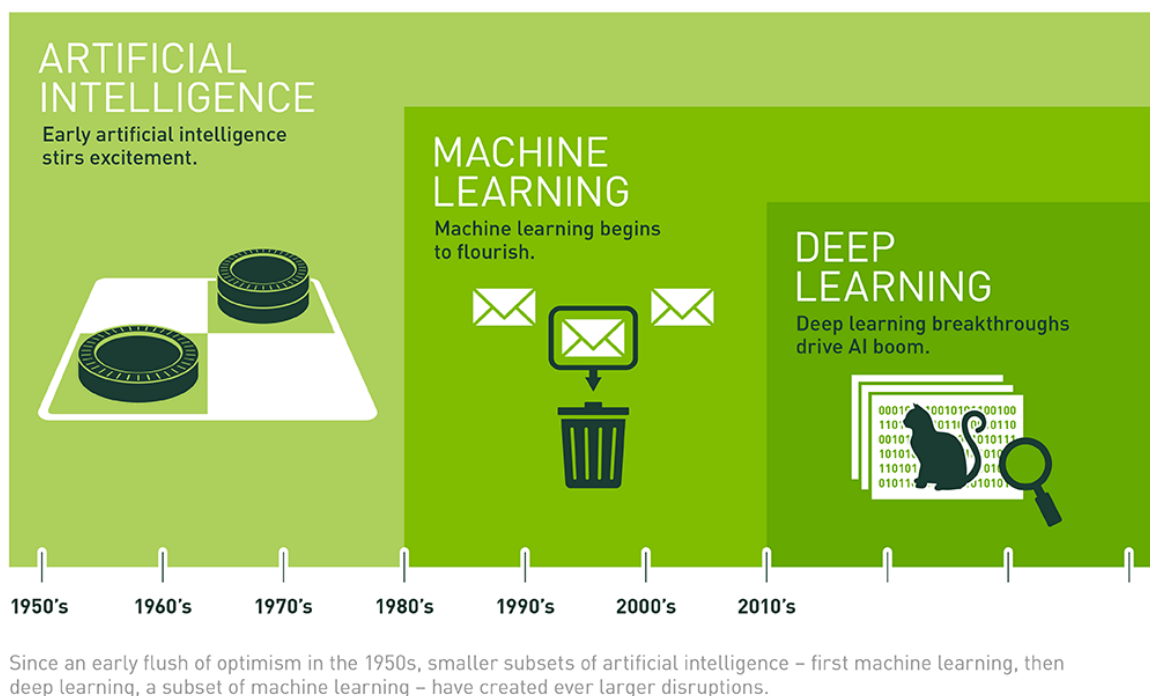
Học máy (*machine learning–ML*) là một tập con của trí tuệ nhân tạo. Nó là một lĩnh vực nhỏ trong khoa học máy tính, có khả năng tự học hỏi dựa trên dữ liệu được đưa vào mà không cần phải được lập trình cụ thể (*Machine Learning is the subfield of computer science, that “gives computers the ability to learn without being explicitly programmed”–Wikipedia*).

Những năm gần đây, sự phát triển của các hệ thống tính toán cùng với lượng dữ liệu khổng lồ được thu thập bởi các hãng công nghệ lớn đã giúp machine learning tiến thêm một bước dài. Một lĩnh vực mới được ra đời được gọi là *học sâu* (*deep learning–DL*). Deep learning đã giúp máy tính thực thi những việc tưởng chừng như không thể vào mười năm trước: phân loại cả ngàn vật thể khác nhau trong các bức ảnh, tự tạo chú thích cho ảnh, bắt chước giọng nói và chữ viết của con người, giao tiếp với con người, chuyển đổi ngôn ngữ, hay thậm chí cả sáng tác văn thơ hay âm nhạc¹.

Mối quan hệ AI-ML-DL

Deep learning là một tập con của machine learning. Machine learning là một tập con của artificial intelligence (xem Hình 0.1).

¹ Đọc thêm: *8 Inspirational Applications of Deep Learning* (<https://goo.gl/Ds3rRy>)



Hình 0.1: Mối quan hệ giữa artificial intelligence, machine learning, và deep learning (Nguồn *What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?* – <https://goo.gl/NNwGCi>).

0.1 Mục đích của cuốn sách

Những phát triển thần kỳ của trí tuệ nhân tạo dẫn đến nhu cầu cao về nhân lực những ngành khoa học dữ liệu, machine learning, và các ngành liên quan trên toàn thế giới cũng như ở Việt Nam trong những năm sắp tới. Đó cũng là động lực để tôi bắt đầu viết blog Machine Learning cơ bản (<https://machinelearningcoban.com>) từ đầu năm 2017. Tính tới thời điểm tôi viết những dòng này, trang blog đã có hơn 650 ngàn lượt ghé thăm. Facebook page Machine Learning cơ bản (<https://goo.gl/wyUEjr>) của blog cũng đã có hơn 10 ngàn lượt likes, Forum Machine Learning cơ bản (<https://goo.gl/gDPTKX>) có gần 8 ngàn thành viên. Trong quá trình viết blog và duy trì các trang Facebook, tôi nhận được rất nhiều những ủng hộ của bạn đọc về tinh thần cũng như vật chất. Ngoài ra, rất nhiều bạn đọc đã khuyến khích tôi tổng hợp những kiến thức trên blog lại thành một cuốn sách cho cộng đồng những người làm machine learning sử dụng tiếng Việt. Những sự ủng hộ và lời động viên đó là động lực lớn cho tôi bắt tay vào thực hiện và hoàn thành cuốn sách này.

Lĩnh vực machine learning và deep learning là cực kỳ rộng lớn và có nhiều nhánh nhỏ. Để đi sâu vào từng nhánh, một cuốn sách chắc chắn không thể bao quát được mọi vấn đề. Mục đích chính của cuốn sách này là cung cấp cho các bạn những khái niệm, kỹ thuật chung và

các thuật toán cơ bản nhất của machine learning. Từ đó, bạn đọc muốn đi sâu vào từng vấn đề cụ thể có thể tìm đọc thêm các tài liệu, cuốn sách, và khoá học liên quan.

Hãy luôn nhớ rằng **đơn giản trước hết**. Khi bắt tay vào giải quyết một bài toán machine learning hay bất cứ bài toán nào, chúng ta nên bắt đầu từ những thuật toán đơn giản nhất. Không nên nghĩ rằng chỉ có những thuật toán phức tạp mới có thể giải quyết được vấn đề. Những thuật toán phức tạp thường yêu cầu độ tính toán cao và nhạy cảm với cách chọn các tham số đầu vào. Thêm vào đó, những thuật toán đơn giản giúp chúng ta sớm có một mô hình tổng quát cho mỗi bài toán. Kết quả của các thuật toán đơn giản, thường được gọi là *baseline*, cũng giúp chúng ta có cái nhìn ban đầu về sự phức tạp của mỗi bài toán. Việc cải thiện kết quả sẽ được dần thực hiện ở các bước sau. Cuốn sách này sẽ giúp các bạn có những cái nhìn đầu tiên và các hướng giải quyết cho các bài toán machine learning. Để có các sản phẩm thực tiễn, chúng ta sẽ phải học hỏi và thực hành thêm rất nhiều.

0.2 Hướng tiếp cận của cuốn sách

Để giải quyết mỗi bài toán machine learning, chúng ta cần chọn một mô hình phù hợp. Mô hình này được mô tả bởi bộ các tham số, có thể lên tới cả triệu tham số, mà chúng ta cần đi tìm. Thông thường, bộ các tham số này được tìm bằng cách giải một bài toán tối ưu.

Khi viết về các thuật toán machine learning, tôi sẽ bắt đầu bằng những ý tưởng trực quan, theo sau bởi một mô hình toán học mô tả ý tưởng đó. Các tham số mô hình được tìm bằng cách tối ưu mô hình toán học đó. Các suy luận toán học và các ví dụ mẫu trên Python ở cuối mỗi bài sẽ giúp bạn đọc hiểu rõ hơn về nguồn gốc, ý nghĩa, và cách sử dụng mỗi thuật toán. Xen kẽ giữa các phần về các thuật toán machine learning, tôi cũng sẽ giới thiệu các kỹ thuật tối ưu cơ bản, với hy vọng giúp bạn đọc hiểu rõ hơn về bản chất của vấn đề.

0.3 Đối tượng của cuốn sách

Cuốn sách được thực hiện hướng đến nhiều nhóm độc giả khác nhau. Nếu bạn không thực sự muốn đi sâu vào phần toán, bạn vẫn có thể tham khảo source code và cách sử dụng các thư viện. Nhưng để sử dụng các thư viện một cách hiệu quả, bạn cũng cần hiểu nguồn gốc của mô hình và ý nghĩa của các tham số. Nếu bạn thực sự muốn tìm hiểu nguồn gốc, ý nghĩa của các thuật toán, bạn có thể học được nhiều điều từ cách xây dựng và tối ưu các mô hình. Phần tổng hợp các kiến thức toán cần thiết trong Phần I sẽ là một nguồn tham khảo súc tích bất cứ khi nào bạn có thắc mắc về các dẫn giải toán học trong sách². Phần VII được dành riêng để nói về tối ưu lồi—một mảng rất quan trọng trong tối ưu, phù hợp với các bạn thực sự muốn đi sâu thêm về tối ưu.

Rất nhiều hình vẽ trong cuốn sách được vẽ dưới dạng vector graphics (độ phân giải rất cao), có thể được dùng trong các bài giảng hoặc thuyết trình. Các kiến thức trong sách cũng được sắp xếp theo thứ tự từ dễ đến khó, vì vậy cuốn sách cũng được hy vọng là một cuốn giáo trình cho các khoá học machine learning tiếng Việt.

² Bạn đọc chưa quen với nhiều khái niệm toán học trong phần này có thể đọc từ Phần II và quay lại bất cứ khi nào bạn gặp khó khăn.

Các dẫn giải toán học được xây dựng phù hợp với chương trình toán phổ thông và đại học ở Việt Nam. Các từ khoá khi được dịch sang tiếng Việt đều dựa trên những tài liệu tôi được học trong nhiều năm học toán tại Việt Nam. Các thuật ngữ tiếng Anh cũng thường xuyên được sử dụng, với hy vọng giúp bạn đọc dần làm quen với các tài liệu tiếng Anh, và giúp các bạn học đại học ở nước ngoài có thể tiếp cận. Phần cuối cùng của sách có mục Index các thuật ngữ quan trọng bằng tiếng Anh và nghĩa tiếng Việt đi kèm nếu tôi tìm được cách dịch phù hợp.

0.4 Yêu cầu về kiến thức

Để có thể bắt đầu đọc cuốn sách này, bạn cần có một kiến thức nhất định về đại số tuyến tính, giải tích ma trận, xác suất thống kê, và kỹ năng lập trình.

Phần I của cuốn sách ôn tập lại các kiến thức toán quan trọng cho machine learning. Bất cứ khi nào bạn đọc gặp khó khăn về toán, bạn được khuyến khích đọc lại các chương trong phần này.

Ngôn ngữ lập trình được sử dụng trong cuốn sách là Python. Lý do tôi sử dụng ngôn ngữ này vì đây là một ngôn ngữ lập trình miễn phí, có thể được cài đặt dễ dàng trên các nền tảng hệ điều hành khác nhau. Quan trọng hơn, có rất nhiều các thư viện hỗ trợ machine learning cũng như deep learning được viết cho Python. Có hai thư viện python chính thường được sử dụng trong cuốn sách là numpy và scikit-learn. Numpy (<http://www.numpy.org/>) là một thư viện phổ biến giúp xử lý các phép toán liên quan đến các mảng nhiều chiều, với các hàm gần gũi với đại số tuyến tính. Nếu bạn đọc chưa quen thuộc với numpy, bạn có thể tham gia một khoá học ngắn miễn phí trên trang web kèm theo cuốn sách này (<https://fundaml.com>). Bạn sẽ được làm quen với cách xử lý các mảng nhiều chiều với nhiều ví dụ và bài tập thực hành trực tiếp trên trình duyệt. Các kỹ thuật xử lý mảng trong cuốn sách này đều được đề cập tại đây. Scikit-learn, hay sklearn, (<http://scikit-learn.org/>) là một thư viện chứa rất nhiều các thuật toán machine learning cơ bản và rất dễ sử dụng. Tài liệu của scikit-learn cũng là một nguồn chất lượng cho các bạn làm machine learning. Scikit-learn sẽ được dùng trong cuốn sách như một cách kiểm chứng lại các kết quả mà chúng ta thực hiện dựa trên suy luận toán học cũng như lập trình thông qua numpy.

Tất nhiên, các thư viện machine learning hiện nay rất phổ biến và có những bạn có thể tạo ra sản phẩm bằng cách chỉ sử dụng những thư viện này mà không cần nhiều kiến thức toán. Tuy nhiên, cuốn sách này không hướng tới việc sử dụng các thư viện sẵn có mà không hiểu bản chất đằng sau của chúng. Việc sử dụng các thư viện cũng yêu cầu những kiến thức nhất định về việc lựa chọn và điều chỉnh tham số mô hình.

0.5 Source code đi kèm

Toàn bộ source code trong cuốn sách có thể được tìm thấy tại https://github.com/tiepvupsu/ebookML_src. Các file có đuôi `.ipynb` là các file chứa code (Jupyter notebook). Các file có đuôi `.pdf`, `.png` là các hình tạo được từ file `.ipynb`.

0.6 Bố cục của cuốn sách

Cuốn sách này được chia thành 8 phần và sẽ tiếp tục được cập nhật:

Phần **I** ôn tập lại cho bạn đọc những kiến thức quan trọng trong đại số tuyến tính, giải tích ma trận, xác suất, và hai phương pháp phổ biến trong việc ước lượng tham số cho các mô hình machine learning thống kê.

Phần **II** giới thiệu các khái niệm cơ bản trong machine learning, kỹ thuật xây dựng vector đặc trưng cho dữ liệu, một mô hình machine learning cơ bản—linear regression, và một hiện tượng cần tránh khi xây dựng các mô hình machine learning.

Phần **III** giúp các bạn làm quen với các mô hình machine learning rất trực quan, không yêu cầu nhiều kiến thức toán phức tạp. Qua đây, bạn đọc sẽ có cái nhìn đầu tiên về việc xây dựng các mô hình machine learning.

Phần **IV** đề cập tới một lớp các thuật toán machine learning phổ biến nhất—neural networks, là nền tảng cho các mô hình deep learning phức tạp hiện nay. Phần này cũng giới thiệu một kỹ thuật cơ bản và hữu dụng trong việc giải quyết các bài toán tối ưu không ràng buộc.

Phần **V** giới thiệu về các kỹ thuật thường dùng trong các hệ thống khuyến nghị sản phẩm.

Phần **VI** giới thiệu các kỹ thuật giảm chiều dữ liệu.

Phần **VII** mang lại cho các bạn một cái nhìn bao quát hơn về tối ưu, đặc biệt là tối ưu lồi. Các bài toán tối ưu lồi có ràng buộc cũng được giới thiệu trong phần này.

Phần **VIII** giới thiệu các thuật toán phân lớp dựa trên ý tưởng của support vector machine.

0.7 Các lưu ý về ký hiệu

Các ký hiệu toán học trong sách được mô tả ở Bảng 0.1 và đầu Chương 1. Các khung với font chữ có chiều rộng các ký tự như nhau được dùng để chứa các đoạn source code.

```
text in a box with constant width represents source codes.
```

Các đoạn ký tự với **constant width**, **deep red**, **'string, dark green'** được dùng để chỉ các biến, hàm số, chuỗi, v.v., trong các đoạn code.

Đóng khung và in nghiêng

Các khái niệm, định nghĩa, định lý, và lưu ý quan trọng được đóng khung và in nghiêng. Ký tự phân cách giữa phần nguyên và phần thập phân của các số thực là dấu chấm, '.', thay vì dấu phẩy, ',', như trong các tài liệu tiếng Việt khác. Cách làm này thống nhất với các tài liệu tiếng Anh và các ngôn ngữ lập trình.

0.8 Tham khảo thêm

Có rất nhiều những cuốn sách, khoá học, website hay về machine learning cũng như deep learning, trong đó, có một số mà tôi muốn đặc biệt nhấn mạnh:

0.8.1 Khoá học

1. Khóa học *Machine Learning* của Andrew Ng trên Coursera (<https://goo.gl/WBwU3K>).
2. Khóa học mới *Deep Learning Specialization* cũng của Andrew Ng (<https://goo.gl/ssXfYN>).
3. Các khoá *CS224n: Natural Language Processing with Deep Learning* (<https://goo.gl/6XTNkH>); *CS231n: Convolutional Neural Networks for Visual Recognition* (<http://cs231n.stanford.edu/>); *CS246: Mining Massive Data Sets* (<https://goo.gl/TEMQ9H>) của Stanford.
4. *Introduction to Computer Science and Programming Using Python* (<https://goo.gl/4nNXvJ>) của MIT.

0.8.2 Sách

1. C. Bishop, *Pattern Recognition and Machine Learning* (<https://goo.gl/pjgqRr>), Springer, 2006 [Bis06].
2. I. Goodfellow *et al.*, *Deep Learning* (<https://goo.gl/sXaGwV>), MIT press, 2016 [GBC16].
3. J. Friedman *et al.*, *The Elements of Statistical Learning* (<https://goo.gl/Qh9EkB>), Springer, 2001 [FHT01].
4. Y. Abu-Mostafa *et al.*, *Learning from data* (<https://goo.gl/SRfNFJ>), AMLBook New York, 2012 [AMMIL12].
5. S. JD Prince, *Computer Vision: Models, Learning, and Inference* (<https://goo.gl/9Fchf3>), Cambridge University Press, 2012 [Pri12].
6. S. Boyd *et al.*, *Convex Optimization* (<https://goo.gl/NomDpC>), Cambridge university press, 2004 [BV04].

Ngoài ra, các website *Machine Learning Mastery* (<https://goo.gl/5DwGbU>), *Pyimage-search* (<https://goo.gl/5DwGbU>). *Kaggle* (<https://www.kaggle.com/>), *Scikit-learn* (<http://scikit-learn.org/>) cũng là các nguồn thông tin rất hữu ích.

0.9 Đóng góp ý kiến

Mọi ý kiến đóng góp, phản hồi, báo lỗi cho nội dung của cuốn sách được tốt hơn đều đáng quý. Các bạn có thể gửi ý kiến tới vuhuutiep@gmail.com hoặc tạo một *issue* mới tại <https://goo.gl/zPYWKV>.

Cuốn sách sẽ tiếp tục được chỉnh sửa và thêm các chương mới *cho tới khi bản sách giấy được ra mắt*. Tất cả các bạn đã đặt ebook sẽ nhận được các bản cập nhật và một bản sách giấy (dự tính vào giữa năm 2018).

0.10 Vấn đề bản quyền

Toàn bộ nội dung trên blog cũng như cuốn sách này (bao gồm cả source code và hình ảnh minh họa) đều thuộc bản quyền của tôi–Vũ Hữu Tiệp.

Tôi rất mong muốn kiến thức của mình tạo ra đến được với nhiều bạn đọc. Tuy nhiên, tôi không ủng hộ bất kỳ một hình thức sao chép không trích nguồn nào. Mọi trích dẫn cần được nêu rõ tên cuốn sách, tên tác giả (Vũ Hữu Tiệp), và link gốc tới blog. Các bài viết trích dẫn quá 25% toàn văn bất kỳ một post nào trên blog hoặc một chương trong cuốn sách này đều không được phép, trừ trường hợp có sự đồng ý của tác giả.

Mọi vấn đề liên quan đến sao chép, phân phát, đăng tải, sử dụng sách và blog, cũng như trao đổi, cộng tác, xin vui lòng liên hệ với tôi tại địa chỉ email vuhuutiep@gmail.com.

0.11 Lời cảm ơn

Trước hết, tôi xin cảm ơn bạn bè trong friend list Facebook của tôi đã nhiệt tình ủng hộ và chia sẻ blog ngay ngày đầu blog được ra mắt. Tôi cũng xin chân thành cảm ơn bạn đọc blog Machine Learning cơ bản và Facebook page Machine Learning cơ bản đã đồng hành cùng tôi trong suốt một năm qua. Không có độc giả, chắc chắn tôi không có đủ động lực viết hơn 30 bài trên blog và rất nhiều các ghi chép nhanh trên Facebook page.

Trong quá trình viết blog, tôi nhận được rất rất nhiều sự ủng hộ của bạn đọc về cả vật chất lẫn tinh thần. Không có những sự ủng hộ đó và những lời động viên viết sách, dự án này sẽ không thể được bắt đầu. Khi tôi đã bắt đầu, số lượng pre-order cuốn sách này tăng lên từng ngày. Tôi thực sự biết ơn các bạn đã pre-order cũng những lời nhắn gửi ấm áp. Quan trọng hơn hết, số lượng sách được đặt trước khi tôi hoàn thành khiến tôi tin rằng sản phẩm mình tạo ra đã mang lại những giá trị nhất định cho cộng đồng. Những điều đó góp phần tôi duy trì tinh thần làm việc và cố gắng hết mình để tạo ra một sản phẩm chất lượng.

Tôi may mắn nhận được những phản hồi tích cực cũng như các góp ý từ các thầy cô trong các trường đại học lớn trong và ngoài nước. Tôi xin được gửi lời cảm ơn tới thầy Phạm Ngọc Nam và cô Nguyễn Việt Hương (ĐH Bách Khoa Hà Nội), thầy Chế Viết Nhật Anh (ĐH Bách Khoa Tp.HCM), thầy Nguyễn Thanh Tùng (ĐH Thủy Lợi), thầy Trần Duy Trác (ĐH

Johns Hopkins), và anh Nguyễn Hồng Lâm (người hướng dẫn trong thời gian tôi thực tập tại U.S. Army Research Lab).

Tôi đặc biệt cảm ơn bạn Nguyễn Hoàng Linh và Hoàng Đức Huy, Đại học Waterloo–Canada, những người bạn đã nhiệt tình giúp tôi xây dựng trang FundaML.com giúp bạn đọc có thể học Python/Numpy trực tiếp trên trình duyệt. Tôi cũng xin cảm ơn bạn Lê Việt Hải–nghiên cứu sinh ngành toán ứng dụng tại Penn State, và Đình Hoàng Phong–kỹ sư phần mềm tại Facebook–đã góp ý sửa đổi rất nhiều điểm về ngôn ngữ và toán trong các bản nháp. Tôi tin rằng cuốn sách đã được sửa đổi rất nhiều so với phiên bản trên blog.

Tôi xin cảm ơn ba người bạn thân–Nguyễn Tiến Cường, Nguyễn Văn Giang, Vũ Đình Quyền–đã luôn đồng viên tôi và đóng góp nhiều phản hồi quý giá cho cuốn sách. Ngoài ra, tôi xin cảm ơn những người bạn thân thiết khác của tôi tại Penn State đã luôn bên cạnh tôi trong thời gian tôi thực hiện dự án, bao gồm gia đình anh Triệu Thanh Quang, gia đình anh Trần Quốc Long, bạn thân (cũng là một blogger) Nguyễn Phương Chi, và các đồng nghiệp John McKay, Tiantong Guo, Hojjat Mousavi, Omar Aldayel, và Mohammad Tofighi trong Phòng nghiên cứu Xử lý Thông tin và Thuật toán (Information Processing and Algorithm Laboratory–iPAL), ĐH bang Pennsylvania.

Cuối cùng và quan trọng nhất, tôi xin cảm ơn gia đình tôi, những người luôn ủng hộ tôi vô điều kiện và hỗ trợ tôi hết mình trong quá trình tôi thực hiện dự án này.

0.12 Bảng các ký hiệu

Các ký hiệu sử dụng trong sách được liệt kê trong Bảng 0.1

Bảng 0.1: Bảng các ký hiệu

Ký hiệu	Ý nghĩa
x, y, N, k	in nghiêng, thường hoặc hoa, là các số vô hướng
\mathbf{x}, \mathbf{y}	in đậm, chữ thường, là các vector
\mathbf{X}, \mathbf{Y}	in đậm, chữ hoa, là các ma trận
\mathbb{R}	tập hợp các số thực
\mathbb{N}	tập hợp các số tự nhiên
\mathbb{C}	tập hợp các số phức
\mathbb{R}^m	tập hợp các vector thực có m phần tử
$\mathbb{R}^{m \times n}$	tập hợp các ma trận thực có m hàng, n cột
\mathbb{S}^n	tập hợp các ma trận vuông đối xứng bậc n
\mathbb{S}_+^n	tập hợp các ma trận nửa xác định dương bậc n
\mathbb{S}_{++}^n	tập hợp các ma trận xác định dương bậc n
\in	phần tử thuộc tập hợp
\exists	tồn tại
\forall	mọi
\triangleq	ký hiệu là/bởi. Ví dụ $a \triangleq f(x)$ nghĩa là “ký hiệu $f(x)$ bởi a ”.
x_i	phần tử thứ i (tính từ 1) của vector \mathbf{x}
$\text{sgn}(x)$	hàm xác định dấu. Bằng 1 nếu $x \geq 0$, bằng -1 nếu $x < 0$.
$\exp(x)$	e^x
$\log(x)$	logarit <i>tự nhiên</i> của số thực dương x
a_{ij}	phần tử hàng thứ i , cột thứ j của ma trận \mathbf{A}
\mathbf{A}^T	chuyển vị của ma trận \mathbf{A}
\mathbf{A}^H	chuyển vị liên hợp (Hermitian) của ma trận phức \mathbf{A}
\mathbf{A}^{-1}	nghịch đảo của ma trận vuông \mathbf{A} , nếu tồn tại
\mathbf{A}^\dagger	giả nghịch đảo của ma trận không nhất thiết vuông \mathbf{A}
\mathbf{A}^{-T}	chuyển vị của nghịch đảo của ma trận \mathbf{A} , nếu tồn tại
$\ \mathbf{x}\ _p$	ℓ_p norm của vector \mathbf{x}
$\ \mathbf{A}\ _F$	Frobenius norm của ma trận \mathbf{A}
$\text{diag}(\mathbf{A})$	đường chéo chính của ma trận \mathbf{A}
$\text{trace}(\mathbf{A})$	trace của ma trận \mathbf{A}
$\det(\mathbf{A})$	định thức của ma trận vuông \mathbf{A}
$\text{rank}(\mathbf{A})$	hạng của ma trận \mathbf{A}
o.w	<i>otherwise</i> – trong các trường hợp còn lại
$\frac{\partial f}{\partial x}$	đạo hàm của hàm số f theo $x \in \mathbb{R}$
$\nabla_{\mathbf{x}} f$	gradient (đạo hàm) của hàm số f theo \mathbf{x} (\mathbf{x} là vector hoặc ma trận)
$\nabla_{\mathbf{x}}^2 f$	đạo hàm bậc hai của hàm số f theo \mathbf{x} , còn được gọi là <i>Hessian</i>
\odot	Hadamard product (elemenwise product). Phép nhân từng phần tử của hai vector hoặc ma trận cùng kích thước.
\propto	tỉ lệ với
v.v.	vân vân

Mục lục

Lời tác giả	i
0.1 Mục đích của cuốn sách	ii
0.2 Hướng tiếp cận của cuốn sách	iii
0.3 Đối tượng của cuốn sách	iii
0.4 Yêu cầu về kiến thức	iv
0.5 Source code đi kèm	iv
0.6 Bố cục của cuốn sách	v
0.7 Các lưu ý về ký hiệu	v
0.8 Tham khảo thêm	vi
0.9 Đóng góp ý kiến	vii
0.10 Vấn đề bản quyền	vii
0.11 Lời cảm ơn	vii
0.12 Bảng các ký hiệu	ix

Phần I Kiến thức toán cơ bản cho machine learning

1 Ôn tập Đại số tuyến tính	12
1.1 Lưu ý về ký hiệu	12

Mục lục	2
1.2 Chuyển vị và Hermitian	12
1.3 Phép nhân hai ma trận	13
1.4 Ma trận đơn vị và ma trận nghịch đảo	14
1.5 Một vài ma trận đặc biệt khác	15
1.6 Định thức	16
1.7 Tổ hợp tuyến tính, không gian sinh	17
1.8 Hạng của ma trận	19
1.9 Hệ trục chuẩn, ma trận trực giao	20
1.10 Biểu diễn vector trong các hệ cơ sở khác nhau	21
1.11 Trị riêng và vector riêng	22
1.12 Chéo hoá ma trận	23
1.13 Ma trận xác định dương	24
1.14 Chuẩn của vector và ma trận	26
2 Giải tích ma trận	30
2.1 Đạo hàm của hàm trả về một số vô hướng	30
2.2 Đạo hàm của hàm trả về một vector	31
2.3 Tính chất quan trọng của đạo hàm	32
2.4 Đạo hàm của các hàm số thường gặp	33
2.5 Bảng các đạo hàm thường gặp	36
2.6 Kiểm tra đạo hàm	36
3 Ôn tập Xác Suất	40
3.1 Xác Suất	40
3.2 Một vài phân phối thường gặp	47

4	Maximum Likelihood và Maximum A Posteriori	52
4.1	Giới thiệu	52
4.2	Maximum likelihood estimation	53
4.3	Maximum a Posteriori	58
4.4	Tóm tắt	62

Phần II Tổng quan về machine learning

5	Các khái niệm cơ bản	64
5.1	Nhiệm vụ, T	64
5.2	Phép đánh giá, P	67
5.3	Kinh nghiệm, E	67
5.4	Hàm mất mát và tham số mô hình	69
6	Giới thiệu về feature engineering	71
6.1	Giới thiệu	71
6.2	Mô hình chung cho các bài toán Machine Learning	72
6.3	Một số ví dụ về Feature Engineering	74
6.4	Transfer Learning cho bài toán phân loại ảnh	79
6.5	Chuẩn hoá vector đặc trưng	81
6.6	Đọc thêm	82
7	Linear regression	83
7.1	Giới thiệu	83
7.2	Xây dựng và tối ưu hàm mất mát	84
7.3	Ví dụ trên Python	86

Mục lục	4
7.4 Thảo luận	89
8 Overfitting	91
8.1 Giới thiệu	91
8.2 Validation	94
8.3 Regularization	96
8.4 Đọc thêm	97
<hr/>	
Phần III Khởi động	
<hr/>	
9 K-nearest neighbors	100
9.1 Giới thiệu	100
9.2 Phân tích toán học	101
9.3 Ví dụ trên cơ sở dữ liệu Iris	105
9.4 Thảo luận	108
10 K-means clustering	110
10.1 Giới thiệu	110
10.2 Phân tích toán học	111
10.3 Ví dụ trên Python	114
10.4 Phân nhóm chữ số viết tay	117
10.5 Tách vật thể trong ảnh	121
10.6 Image Compression (nén ảnh và nén dữ liệu nói chung)	122
10.7 Thảo luận	123
11 Naive Bayes classifier	127
11.1 Naive Bayes classifier	127

11.2 Các phân phối thường dùng trong NBC	128
11.3 Ví dụ	130
11.4 Thảo luận	137

Phần IV Neural networks

12 Gradient descent	140
12.1 Giới thiệu	140
12.2 GD cho hàm một biến	141
12.3 GD cho hàm nhiều biến	145
12.4 GD với momentum	148
12.5 Nesterov accelerated gradient	151
12.6 Stochastic gradient descent	152
12.7 Thảo luận	155
13 Perceptron learning algorithm	156
13.1 Giới thiệu	156
13.2 Thuật toán perceptron	157
13.3 Ví dụ và minh họa trên Python	160
13.4 Mô hình neural network đầu tiên	162
13.5 Thảo Luận	163
14 Logistic regression	165
14.1 Giới thiệu	165
14.2 Hàm mất mát và phương pháp tối ưu	167
14.3 Triển khai thuật toán trên Python	169

14.4	Tính chất của logistic regression	172
14.5	Bài toán phân biệt hai chữ số viết tay	174
14.6	Bộ phân lớp nhị phân cho bài toán phân lớp đa lớp	175
14.7	Thảo luận	177
15	Softmax regression	180
15.1	Giới thiệu	180
15.2	Softmax function	181
15.3	Hàm mất mát và phương pháp tối ưu	184
15.4	Ví dụ trên Python	189
15.5	Thảo luận	191
16	Multilayer neural network và backpropagation	193
16.1	Giới thiệu	193
16.2	Các ký hiệu và khái niệm	196
16.3	Activation function–Hàm kích hoạt	197
16.4	Backpropagation	200
16.5	Ví dụ trên Python	204
16.6	Tránh overfitting cho neural network bằng weight decay	209
16.7	Đọc thêm	211

Phần V Recommendation systems–Hệ thống khuyến nghị

17	Content-based recommendation system	214
17.1	Giới thiệu	214
17.2	Utility matrix	215

17.3	Content-based recommendation	217
17.4	Bài toán với cơ sở dữ liệu MovieLens 100k	220
17.5	Thảo luận	224
18	Neighborhood-based collaborative filtering	225
18.1	Giới thiệu	225
18.2	User-user collaborative filtering	226
18.3	Item-item collaborative filtering	230
18.4	Lập trình trên Python	232
18.5	Thảo luận	235
19	Matrix factorization collaborative filtering	236
19.1	Giới thiệu	236
19.2	Xây dựng và tối ưu hàm mất mát	238
19.3	Lập trình Python	240
19.4	Thảo luận	243

Phần VI Dimensionality reduction–Giảm chiều dữ liệu

20	Singular value decomposition	246
20.1	Giới thiệu	246
20.2	Singular value decomposition	247
20.3	SVD cho image compression	252
20.4	Thảo luận	253
21	Principal component analysis	254
21.1	Principal component analysis	254

21.2 Các bước thực hiện PCA	259
21.3 Mối quan hệ giữa PCA và SVD	259
21.4 Làm thế nào để chọn số chiều của dữ liệu mới	261
21.5 Lưu ý về tính PCA trong các bài toán thực tế	262
21.6 Một vài ứng dụng của PCA	263
21.7 Thảo luận	266
22 Linear discriminant analysis	268
22.1 Giới thiệu	268
22.2 LDA cho bài toán phân lớp nhị phân	270
22.3 LDA cho bài toán phân lớp nhiều lớp	273
22.4 Ví dụ trên Python	276
22.5 Thảo luận	278

Phần VII Convex optimization–Tối ưu lồi

23 Tập lồi và hàm lồi	282
23.1 Giới thiệu	282
23.2 Tập lồi – Convex sets	283
23.3 Convex functions	288
23.4 Tóm tắt	298
24 Bài toán tối ưu lồi	299
24.1 Giới thiệu	299
24.2 Nhắc lại bài toán tối ưu	303
24.3 Bài toán tối ưu lồi	305

24.4	Linear programming	307
24.5	Quadratic programming	310
24.6	Geometric Programming	313
24.7	Tóm tắt	316
25	Duality	317
25.1	Giới thiệu	317
25.2	Hàm đối ngẫu Lagrange	318
25.3	Bài toán đối ngẫu Lagrange	321
25.4	Các điều kiện tối ưu	323
25.5	Tóm tắt	325

Phần VIII Support vector machines

26	Support vector machine	328
26.1	Giới thiệu	328
26.2	Xây dựng bài toán tối ưu cho SVM	330
26.3	Bài toán đối ngẫu của SVM	332
26.4	Lập trình tìm nghiệm cho SVM	336
26.5	Tóm tắt và thảo luận	338
27	Soft-margin support vector machine	339
27.1	Giới thiệu	339
27.2	Phân tích toán học	340
27.3	Bài toán đối ngẫu Lagrange	342
27.4	Bài toán tối ưu không ràng buộc cho <i>soft-margin SVM</i>	345

Mục lục	10
27.5 Lập trình với <i>soft-margin SVM</i>	349
27.6 Tóm tắt và thảo luận	353
28 Kernel support vector machine	355
28.1 Giới thiệu	355
28.2 Cơ sở toán học	357
28.3 Hàm số kernel	359
28.4 Ví dụ minh họa	361
28.5 Tóm tắt	363
29 Multi-class support vector machine	364
29.1 Giới thiệu	364
29.2 Xây dựng hàm mất mát	368
29.3 Tính toán hàm mất mát và đạo hàm của nó	371
29.4 Thảo luận	378
A Phương pháp nhân tử Lagrange	379
Tài liệu tham khảo	383
Index	386

Kiến thức toán cơ bản cho machine learning

Ôn tập Đại số tuyến tính

1.1 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 in nghiêng, 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, và $\mathbf{x} = [x_1; x_2; \dots; x_n]$ 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. Nếu không giải thích gì thêm, một chữ cái viết thường in đậm được hiểu là một vector cột.

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. Phần tử ở hàng thứ i , cột thứ j được ký hiệu là x_{ij} .

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.

1.2 Chuyển vị và Hermitian

Một toán tử quan trọng của ma trận hay vector là toán tử *chuyển vị* (*transpose*).

Cho $\mathbf{A} \in \mathbb{R}^{m \times n}$, ta nói $\mathbf{B} \in \mathbb{R}^{n \times m}$ là chuyển vị của \mathbf{A} nếu $b_{ij} = a_{ji}$, $\forall 1 \leq i \leq n, 1 \leq j \leq m$.

Một cách ngắn gọn, chuyển vị của một ma trận là một ma trận nhận được từ ma trận cũ thông qua phép phản xạ gương qua đường chéo chính của ma trận ban đầu. Toán tử chuyển

vị thường được ký hiệu bởi chữ T , t hoặc ký tự \top . Trong cuốn sách này, chúng ta sẽ sử dụng chữ cái T . Ví dụ, chuyển vị của một vector \mathbf{x} được ký hiệu là \mathbf{x}^T ; chuyển vị của một ma trận \mathbf{A} được ký hiệu là \mathbf{A}^T . Cụ thể:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \Rightarrow \mathbf{x}^T = [x_1 \ x_2 \ \dots \ x_m]; \quad \mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \ddots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \Rightarrow \mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \dots & \dots & \ddots & \dots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$

Nếu $\mathbf{A} \in \mathbb{R}^{m \times n}$ thì $\mathbf{A}^T \in \mathbb{R}^{n \times m}$. Nếu $\mathbf{A}^T = \mathbf{A}$, ta nói \mathbf{A} là một *ma trận đối xứng* (*symmetric matrix*).

Trong trường hợp vector hay ma trận có các phần tử là số phức, việc lấy chuyển vị thường đi kèm với việc lấy liên hợp phức. Tức là ngoài việc đổi vị trí của các phần tử, ta còn lấy liên hợp phức của các phần tử đó. Tên gọi của phép toán chuyển vị và lấy liên hợp này còn được gọi là *chuyển vị liên hợp* (*conjugate transpose*), và thường được ký hiệu bằng chữ H thay cho chữ T . Chuyển vị liên hợp của một ma trận \mathbf{A} được ký hiệu là \mathbf{A}^H (cũng được đọc là \mathbf{A} Hermitian).

Cho $\mathbf{A} \in \mathbb{C}^{m \times n}$, ta nói $\mathbf{B} \in \mathbb{C}^{n \times m}$ là chuyển vị liên hợp của \mathbf{A} nếu $b_{ij} = \bar{a}_{ji}$, $\forall 1 \leq i \leq n, 1 \leq j \leq m$, trong đó \bar{a} là liên hiệp phức của a .

Ví dụ:

$$\mathbf{A} = \begin{bmatrix} 1 + 2i & 3 - 4i \\ i & 2 \end{bmatrix} \Rightarrow \mathbf{A}^H = \begin{bmatrix} 1 - 2i & -i \\ 3 + 4i & 2 \end{bmatrix}; \quad \mathbf{x} = \begin{bmatrix} 2 + 3i \\ 2i \end{bmatrix} \Rightarrow \mathbf{x}^H = [2 - 3i \quad -2i] \quad (1.1)$$

Nếu \mathbf{A}, \mathbf{x} là các ma trận và vector thực thì $\mathbf{A}^H = \mathbf{A}^T, \mathbf{x}^H = \mathbf{x}^T$.

Nếu chuyển vị liên hợp của một ma trận phức bằng với chính nó, $\mathbf{A}^H = \mathbf{A}$, thì ta nói ma trận đó là *Hermitian*.

1.3 Phép nhân hai ma trận

Cho hai ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times p}$, tích của hai ma trận được ký hiệu là $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times p}$ trong đó phần tử ở hàng thứ i , cột thứ j của ma trận kết quả được tính bởi:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, \quad \forall 1 \leq i \leq m, 1 \leq j \leq p \quad (1.2)$$

Để nhân được hai ma trận, số cột của ma trận thứ nhất phải bằng số hàng của ma trận thứ hai. Trong ví dụ trên, chúng đều bằng n .

Một vài tính chất của phép nhân hai ma trận (giả sử kích thước các ma trận là phù hợp để các phép nhân ma trận tồn tại):

1. Phép nhân ma trận không có tính chất giao hoán. Thông thường (không phải luôn luôn), $\mathbf{AB} \neq \mathbf{BA}$. Thậm chí, trong nhiều trường hợp, các phép tính này không tồn tại vì kích thước các ma trận lệch nhau.
2. Phép nhân ma trận có tính chất kết hợp: $\mathbf{ABC} = (\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$
3. Phép nhân ma trận có tính chất phân phối đối với phép cộng: $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$.
4. Chuyển vị của một tích bằng tích các chuyển vị theo thứ tự ngược lại. Điều tương tự xảy ra với Hermitian của một tích:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T; \quad (\mathbf{AB})^H = \mathbf{B}^H \mathbf{A}^H \quad (1.3)$$

Theo định nghĩa trên, bằng cách coi vector là một trường hợp đặc biệt của ma trận, tích vô hướng của hai vector (*inner product*) $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ được định nghĩa là:

$$\mathbf{x}^T \mathbf{y} = \mathbf{y}^T \mathbf{x} = \sum_{i=1}^n x_i y_i \quad (1.4)$$

Chú ý, $\mathbf{x}^H \mathbf{y} = (\mathbf{y}^H \mathbf{x})^H = \mathbf{y}^H \mathbf{x}$. Chúng bằng nhau khi và chỉ khi chúng là các số thực. Nếu tích vô hướng của hai vector khác không bằng không, hai vector đó vuông góc với nhau.

$\mathbf{x}^H \mathbf{x} \geq 0, \forall \mathbf{x} \in \mathbb{C}^n$ vì tích của một số phức với liên hiệp của nó luôn là một số không âm.

Phép nhân của một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ với một vector $\mathbf{x} \in \mathbb{R}^n$ là một vector $\mathbf{b} \in \mathbb{R}^m$:

$$\mathbf{Ax} = \mathbf{b}, \quad \text{với } b_i = \mathbf{A}_{:,i} \mathbf{x} \quad (1.5)$$

với $\mathbf{A}_{:,i}$ là vector hàng thứ i của \mathbf{A} .

Ngoài ra, một phép nhân khác được gọi là *Hadamard* (hay *element-wise*) hay được sử dụng trong Machine Learning. Tích Hadamard của hai ma trận cùng kích thước $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times n}$, ký hiệu là $\mathbf{C} = \mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{m \times n}$, trong đó:

$$c_{ij} = a_{ij} b_{ij} \quad (1.6)$$

1.4 Ma trận đơn vị và ma trận nghịch đảo

1.4.1 Ma trận đơn vị

Đường chéo chính của một ma trận là tập hợp các điểm có chỉ số hàng và cột là như nhau. Cách định nghĩa này cũng có thể được định nghĩa cho một ma trận không vuông. Cụ thể, nếu $\mathbf{A} \in \mathbb{R}^{m \times n}$ thì đường chéo chính của \mathbf{A} bao gồm $\{a_{11}, a_{22}, \dots, a_{pp}\}$, trong đó $p = \min\{m, n\}$.

Một ma trận đơn vị bậc n là một ma trận đặc biệt trong $\mathbb{R}^{n \times n}$ với các phần tử trên đường chéo chính bằng 1, các phần tử còn lại bằng 0. Ma trận đơn vị thường được ký hiệu là \mathbf{I}

(identity matrix). Nếu làm việc với nhiều ma trận đơn vị với bậc khác nhau, ta thường ký hiệu \mathbf{I}_n cho ma trận đơn vị bậc n . Dưới đây là ma trận đơn vị bậc 3 và bậc 4:

$$\mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{I}_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.7)$$

Ma trận đơn vị có tính chất đặc biệt trong phép nhân. Nếu $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times m}$ và \mathbf{I} là ma trận đơn vị bậc n , ta có: $\mathbf{AI} = \mathbf{A}$, $\mathbf{IB} = \mathbf{B}$.

Với mọi vector $\mathbf{x} \in \mathbb{R}^n$, ta có $\mathbf{I}_n \mathbf{x} = \mathbf{x}$.

1.4.2 Ma trận nghịch đảo

Cho một ma trận vuông $\mathbf{A} \in \mathbb{R}^{n \times n}$, nếu tồn tại ma trận vuông $\mathbf{B} \in \mathbb{R}^{n \times n}$ sao cho $\mathbf{AB} = \mathbf{I}_n$, thì ta nói \mathbf{A} là *khả nghịch* (*invertible*, *nonsingular* hoặc *nondegenerate*), và \mathbf{B} được gọi là *ma trận nghịch đảo* (*inverse matrix*) của \mathbf{A} . Nếu không tồn tại ma trận \mathbf{B} thỏa mãn điều kiện trên, ta nói rằng ma trận \mathbf{A} là không khả nghịch (*singular* hoặc *degenerate*).

Nếu \mathbf{A} là khả nghịch, ma trận nghịch đảo của nó thường được ký hiệu là \mathbf{A}^{-1} . Ta cũng có:

$$\mathbf{A}^{-1} \mathbf{A} = \mathbf{AA}^{-1} = \mathbf{I} \quad (1.8)$$

Ma trận nghịch đảo thường được sử dụng để giải hệ phương trình tuyến tính. Giả sử rằng $\mathbf{A} \in \mathbb{R}^{n \times n}$ là một ma trận khả nghịch và một vector bất kỳ $\mathbf{b} \in \mathbb{R}^n$. Khi đó, phương trình:

$$\mathbf{Ax} = \mathbf{b} \quad (1.9)$$

có nghiệm duy nhất là $\mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$. Thật vậy, nhân bên trái cả hai vế của phương trình với \mathbf{A}^{-1} , ta có $\mathbf{Ax} = \mathbf{b} \Leftrightarrow \mathbf{A}^{-1} \mathbf{Ax} = \mathbf{A}^{-1} \mathbf{b} \Leftrightarrow \mathbf{x} = \mathbf{A}^{-1} \mathbf{b}$.

Nếu \mathbf{A} không khả nghịch, thậm chí không vuông, phương trình tuyến tính (1.9) có thể không có nghiệm hoặc có vô số nghiệm.

Giả sử các ma trận vuông \mathbf{A}, \mathbf{B} là khả nghịch, khi đó tích của chúng cũng khả nghịch, và $(\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}$. Quy tắc này cũng khá giống với cách tính ma trận chuyển vị của tích các ma trận.

1.5 Một vài ma trận đặc biệt khác

1.5.1 Ma trận đường chéo

Ma trận đường chéo (*diagonal matrix*) là ma trận chỉ có các thành phần trên đường chéo chính là khác không. Định nghĩa này cũng có thể được áp dụng lên các ma trận không vuông. Ma trận không (tất cả các phần tử bằng 0) và đơn vị là các ma trận đường chéo. Một vài ví

dụ về các ma trận đường chéo $[1]$, $\begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix}$, $\begin{bmatrix} -1 & 0 \\ 0 & 2 \\ 0 & 0 \end{bmatrix}$.

Với các ma trận đường chéo vuông, thay vì viết cả ma trận, ta có thể chỉ liệt kê các thành phần trên đường chéo. Ví dụ, một ma trận đường chéo vuông $\mathbf{A} \in \mathbb{R}^{m \times m}$ có thể được ký hiệu là $\text{diag}(a_{11}, a_{22}, \dots, a_{mm})$ với a_{ii} là phần tử hàng thứ i , cột thứ i của ma trận \mathbf{A} .

Tích, tổng của hai ma trận đường chéo vuông cùng bậc là một ma trận đường chéo. Một ma trận đường chéo vuông là khả nghịch nếu và chỉ nếu mọi phần tử trên đường chéo chính là khác không. Nghịch đảo của một ma trận đường chéo khả nghịch cũng là một ma trận đường chéo. Cụ thể hơn, $(\text{diag}(a_1, a_2, \dots, a_n))^{-1} = \text{diag}(a_1^{-1}, a_2^{-1}, \dots, a_n^{-1})$.

1.5.2 Ma trận tam giác

Một ma trận vuông được gọi là *ma trận tam giác trên* (*upper triangular matrix*) nếu tất cả các thành phần nằm phía dưới đường chéo chính bằng 0. Tương tự, một ma trận vuông được gọi là *ma trận tam giác dưới* (*lower triangular matrix*) nếu tất cả các thành phần nằm phía trên đường chéo chính bằng 0.

Các hệ phương trình tuyến tính mà ma trận hệ số có dạng tam giác thường được quan tâm vì chúng có thể được giải với chi phí tính toán thấp (*low computational cost*). Xét hệ:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1,n-1}x_{n-1} + a_{1n}x_n = b_1 \\ a_{22}x_2 + \dots + a_{2,n-1}x_{n-1} + a_{2n}x_n = b_2 \\ \dots \quad \dots \quad \dots \quad \dots \\ a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = b_{n-1} \\ a_{nn}x_n = b_n \end{cases} \quad (1.10)$$

Hệ này có thể được viết gọn dưới dạng $\mathbf{Ax} = \mathbf{b}$ với \mathbf{A} là một ma trận tam giác trên. Nhận thấy rằng phương trình này có thể giải mà không cần tính ma trận nghịch đảo \mathbf{A}^{-1} (quá trình tính ma trận nghịch đảo thường tốn khá nhiều thời gian), thay vào đó, ta có thể giải x_n dựa vào phương trình cuối cùng. Sau khi có x_n , ta có thể thay nó vào phương trình gần cuối để suy ra x_{n-1} . Tiếp tục quá trình này, ta sẽ có nghiệm cuối cùng \mathbf{x} . Quá trình giải từ cuối lên đầu và thay toàn bộ các thành phần đã tìm được vào phương trình hiện tại được gọi là *back substitution*. Nếu ma trận hệ số là một ma trận tam giác dưới, hệ phương trình có thể được giải bằng một quá trình ngược lại – lần lượt tính x_1 rồi x_2, \dots, x_n , quá trình này được gọi là *forward substitution*.

1.6 Định thức

1.6.1 Định nghĩa

Định thức của một ma trận vuông \mathbf{A} được ký hiệu là $\det(\mathbf{A})$ hoặc $\det \mathbf{A}$. Có nhiều cách định nghĩa khác nhau của *định thức* (determinant). Chúng ta sẽ sử dụng cách định nghĩa dựa trên quy nạp theo bậc n của ma trận.

Với $n = 1$, $\det(\mathbf{A})$ chính là phần tử duy nhất của ma trận đó.

Với một ma trận vuông bậc $n > 1$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \ddots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \Rightarrow \det(\mathbf{A}) = \sum_{j=1}^n (-1)^{i+j} a_{ij} \det(\mathbf{A}_{ij}) \quad (1.11)$$

Trong đó $1 \leq i \leq n$ bất kỳ và \mathbf{A}_{ij} là phần bù đại số của \mathbf{A} ứng với phần tử ở hàng i , cột j . Phần bù đại số này là một ma trận con của \mathbf{A} nhận được từ \mathbf{A} bằng cách xoá hàng thứ i và cột thứ j của nó. Đây chính là cách tính định thức dựa trên cách khai triển hàng thứ i của ma trận¹.

1.6.2 Tính chất

1. $\det(\mathbf{A}) = \det(\mathbf{A}^T)$: Một ma trận bất kỳ và chuyển vị của nó có định thức như nhau.
2. Định thức của một ma trận đường chéo (và vuông) bằng tích các phần tử trên đường chéo chính. Nói cách khác, nếu $\mathbf{A} = \text{diag}(a_1, a_2, \dots, a_n)$, thì $\det(\mathbf{A}) = a_1 a_2 \dots a_n$.
3. Định thức của một ma trận đơn vị bằng 1.
4. Định thức của một tích bằng tích các định thức.

$$\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B}) \quad (1.12)$$

với \mathbf{A}, \mathbf{B} là hai ma trận vuông cùng chiều.

5. Nếu một ma trận có một hàng hoặc một cột là một vector $\mathbf{0}$, thì định thức của nó bằng 0.
6. Một ma trận là khả nghịch khi và chỉ khi định thức của nó khác 0.
7. Nếu một ma trận khả nghịch, định thức của ma trận nghịch đảo của nó bằng nghịch đảo định thức của nó.

$$\det(\mathbf{A}^{-1}) = \frac{1}{\det(\mathbf{A})} \text{ vì } \det(\mathbf{A}) \det(\mathbf{A}^{-1}) = \det(\mathbf{AA}^{-1}) = \det(\mathbf{I}) = 1. \quad (1.13)$$

1.7 Tổ hợp tuyến tính, không gian sinh

1.7.1 Tổ hợp tuyến tính

Cho các vector khác không $\mathbf{a}_1, \dots, \mathbf{a}_n \in \mathbb{R}^m$ và các số thực $x_1, \dots, x_n \in \mathbb{R}$, vector:

$$\mathbf{b} = x_1 \mathbf{a}_1 + x_2 \mathbf{a}_2 + \dots + x_n \mathbf{a}_n \quad (1.14)$$

được gọi là một tổ hợp tuyến tính (linear combination) của $\mathbf{a}_1, \dots, \mathbf{a}_n$. Xét ma trận $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n] \in \mathbb{R}^{m \times n}$ và $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$, biểu thức (1.14) có thể được viết lại thành $\mathbf{b} = \mathbf{Ax}$. Ta có thể nói rằng \mathbf{b} là một tổ hợp tuyến tính các cột của \mathbf{A} .

¹ Việc ghi nhớ định nghĩa này không thực sự quan trọng bằng việc ta cần nhớ một vài tính chất của nó.

Tập hợp tất cả các vector có thể biểu diễn được dưới dạng một tổ hợp tuyến tính của các cột của một ma trận được gọi là *không gian sinh* (*span space*, hoặc gọn là *span*) các cột của ma trận đó. Không gian sinh của một hệ các vector thường được ký hiệu là $\text{span}(\mathbf{a}_1, \dots, \mathbf{a}_n)$. Nếu phương trình:

$$\mathbf{0} = x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \dots + x_n\mathbf{a}_n \quad (1.15)$$

có nghiệm duy nhất $x_1 = x_2 = \dots = x_n = 0$, ta nói rằng hệ $\{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$ là một hệ *độc lập tuyến tính* (linear independence). Ngược lại, Nếu tồn tại $x_i \neq 0$ sao cho phương trình trên thoả mãn, ta nói rằng đó là một hệ *phụ thuộc tuyến tính* (linear dependence).

1.7.2 Tính chất

1. Một hệ là phụ thuộc tuyến tính nếu và chỉ nếu tồn tại một vector trong hệ đó là tổ hợp tuyến tính của các vector còn lại. Thật vậy, giả sử phương trình (1.15) có nghiệm khác không. Giả sử hệ số khác không là x_i , ta sẽ có:

$$\mathbf{a}_i = \frac{-x_1}{x_i}\mathbf{a}_1 + \dots + \frac{-x_{i-1}}{x_i}\mathbf{a}_{i-1} + \frac{-x_{i+1}}{x_i}\mathbf{a}_{i+1} + \dots + \frac{-x_n}{x_i}\mathbf{a}_n \quad (1.16)$$

tức \mathbf{a}_i là một tổ hợp tuyến tính của các vector còn lại.

2. Tập con khác rỗng của một hệ độc lập tuyến tính là một hệ độc lập tuyến tính.
3. Tập hợp các cột của một ma trận khả nghịch tạo thành một hệ độc lập tuyến tính.

Giả sử ma trận \mathbf{A} khả nghịch, phương trình $\mathbf{Ax} = \mathbf{0}$ có nghiệm duy nhất $\mathbf{x} = \mathbf{A}^{-1}\mathbf{0} = \mathbf{0}$. Vì vậy, các cột của \mathbf{A} tạo thành một hệ độc lập tuyến tính.

4. Nếu \mathbf{A} là một ma trận cao (*tall matrix*), tức số hàng lớn hơn số cột, $m > n$, thì tồn tại vector \mathbf{b} sao cho $\mathbf{Ax} = \mathbf{b}$ vô nghiệm.

Việc này có thể dễ hình dung trong không gian ba chiều. Không gian sinh của một vector là một đường thẳng, không gian sinh của hai vector độc lập tuyến tính là một mặt phẳng, tức chỉ biểu diễn được các vector nằm trong mặt phẳng đó.

Ta cũng có thể chứng minh tính chất này bằng phản chứng. Giả sử mọi vector trong không gian m chiều đều nằm trong không gian sinh của một hệ $n < m$ vector là các cột của một ma trận \mathbf{A} . Xét các cột của ma trận đơn vị bậc m . Vì mọi cột của ma trận này đều có thể biểu diễn dưới dạng một tổ hợp tuyến tính của n vector đã cho nên phương trình $\mathbf{AX} = \mathbf{I}$ có nghiệm. Nếu ta thêm các vào các cột bằng 0 và các hàng bằng 0 vào \mathbf{A} và \mathbf{X} để được các ma trận vuông, ta sẽ có $[\mathbf{A} \ \mathbf{0}] \begin{bmatrix} \mathbf{X} \\ \mathbf{0} \end{bmatrix} = \mathbf{I}$. Việc này chỉ ra rằng $[\mathbf{A} \ \mathbf{0}]$ là một ma trận khả nghịch trong khi nó có các cột bằng 0. Đây là một điều vô lý vì theo tính chất của định thức, định thức của $[\mathbf{A} \ \mathbf{0}]$ bằng 0.

5. Nếu $n > m$, thì n vector bất kỳ trong không gian m chiều tạo thành một hệ phụ thuộc tuyến tính. Xin được bỏ qua phần chứng minh.

1.7.3 Cơ sở của một không gian

Một hệ các vector $\{\mathbf{a}_1, \dots, \mathbf{a}_n\}$ trong không gian vector m chiều $V = \mathbb{R}^m$ được gọi là một cơ sở (basic) nếu hai điều kiện sau được thoả mãn:

1. $V \equiv \text{span}(\mathbf{a}_1, \dots, \mathbf{a}_n)$
2. $\{\mathbf{a}_1, \dots, \mathbf{a}_n\}$ là một hệ độc lập tuyến tính.

Khi đó, mọi vector $\mathbf{b} \in V$ đều có thể biểu diễn *duy nhất* dưới dạng một tổ hợp tuyến tính của các \mathbf{a}_i .

Từ hai tính chất cuối ở mục trước, ta có thể suy ra rằng $m = n$.

1.7.4 Range và Null space

Với mỗi ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$, có hai không gian con quan trọng ứng với ma trận này.

1. Range của \mathbf{A} . Range của \mathbf{A} , được định nghĩa là:

$$\mathcal{R}(\mathbf{A}) = \{\mathbf{y} \in \mathbb{R}^m : \exists \mathbf{x} \in \mathbb{R}^n, \mathbf{A}\mathbf{x} = \mathbf{y}\} \quad (1.17)$$

Nói cách khác, $\mathcal{R}(\mathbf{A})$ là tập hợp các điểm là tổ hợp tuyến tính của các cột của \mathbf{A} , hay chính là không gian sinh (span) của các cột của \mathbf{A} . $\mathcal{R}(\mathbf{A})$ là một không gian con của \mathbb{R}^m với số chiều chính bằng số lượng lớn nhất các cột của \mathbf{A} độc lập tuyến tính.

2. Null của \mathbf{A} , ký hiệu là $\mathcal{N}(\mathbf{A})$, được định nghĩa là:

$$\mathcal{N}(\mathbf{A}) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{A}\mathbf{x} = \mathbf{0}\} \quad (1.18)$$

Mỗi vector trong $\mathcal{N}(\mathbf{A})$ chính là một bộ các hệ số làm cho tổ hợp tuyến tính các cột của \mathbf{A} tạo thành một vector 0. $\mathcal{N}(\mathbf{A})$ có thể được chứng minh là một không gian con trong \mathbb{R}^n . Khi các cột của \mathbf{A} là độc lập tuyến tính, theo định nghĩa, $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$ (chỉ gồm vector $\mathbf{0}$).

$\mathcal{R}(\mathbf{A})$ và $\mathcal{N}(\mathbf{A})$ là các không gian con vector với số chiều lần lượt là $\dim(\mathcal{R}(\mathbf{A}))$ và $\dim(\mathcal{N}(\mathbf{A}))$, ta có tính chất quan trọng sau đây:

$$\dim(\mathcal{R}(\mathbf{A})) + \dim(\mathcal{N}(\mathbf{A})) = n \quad (1.19)$$

1.8 Hạng của ma trận

Xét một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$. *Hạng* (rank) của ma trận này, ký hiệu là $\text{rank}(\mathbf{A})$, được định nghĩa là số lượng lớn nhất các cột của nó tạo thành một hệ độc lập tuyến tính.

Các tính chất quan trọng của hạng:

1. Một ma trận có hạng bằng 0 khi và chỉ khi nó là ma trận 0.
2. $\text{rank}(\mathbf{A}) = \text{rank}(\mathbf{A}^T)$. Hạng của một ma trận bằng hạng của ma trận chuyển vị. Nói cách khác, số lượng lớn nhất các cột độc lập tuyến tính của một ma trận bằng với số lượng lớn nhất các hàng độc lập tuyến tính của ma trận đó. Từ đây ta suy ra:
3. Nếu $\mathbf{A} \in \mathbb{R}^{m \times n}$, thì $\text{rank}(\mathbf{A}) \leq \min(m, n)$ vì theo định nghĩa, hạng của một ma trận không thể lớn hơn số hàng hoặc số cột của nó.
4. $\text{rank}(\mathbf{AB}) \leq \min(\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B}))$
5. $\text{rank}(\mathbf{A} + \mathbf{B}) \leq \text{rank}(\mathbf{A}) + \text{rank}(\mathbf{B})$. Điều này chỉ ra rằng một ma trận có hạng bằng k không được biểu diễn dưới dạng ít hơn k ma trận có hạng bằng 1. Đến bài Singular Value Decomposition, chúng ta sẽ thấy rằng một ma trận có hạng bằng k có thể biểu diễn được dưới dạng đúng k ma trận có hạng bằng 1.
6. Bất đẳng thức Sylvester về hạng: Nếu $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times k}$, thì

$$\text{rank}(\mathbf{A}) + \text{rank}(\mathbf{B}) - n \leq \text{rank}(\mathbf{AB})$$

Xét một ma trận vuông $\mathbf{A} \in \mathbb{R}^{n \times n}$, hai điều kiện bất kỳ dưới đây là tương đương:

1. \mathbf{A} là một ma trận khả nghịch.
2. Các cột của \mathbf{A} tạo thành một cơ sở trong không gian n chiều.
3. $\det(\mathbf{A}) \neq 0$.
4. $\text{rank}(\mathbf{A}) = n$

1.9 Hệ trực chuẩn, ma trận trực giao**1.9.1 Định nghĩa**

Một hệ cơ sở $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$ được gọi là *trực giao* (*orthogonal*) nếu mỗi vector là khác 0 và tích của hai vector khác nhau bất kỳ bằng 0:

$$\mathbf{u}_i \neq \mathbf{0}; \quad \mathbf{u}_i^T \mathbf{u}_j = 0 \quad \forall 1 \leq i \neq j \leq m \quad (1.20)$$

Một hệ cơ sở $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$ được gọi là *trực chuẩn* (*orthonormal*) nếu nó là một hệ *trực giao* và độ dài Euclidean (xem thêm phần ℓ_2 norm) của mỗi vector bằng 1:

$$\mathbf{u}_i^T \mathbf{u}_j = \begin{cases} 1 & \text{nếu } i = j \\ 0 & \text{o.w.} \end{cases} \quad (1.21)$$

(o.w. là cách viết ngắn gọn của *trong các trường hợp còn lại* (viết tắt của *otherwise*).)

Gọi $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m]$ với $\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m \in \mathbb{R}^m\}$ là *trực chuẩn*, từ (1.21) có thể suy ra:

$$\mathbf{U}\mathbf{U}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I} \quad (1.22)$$

trong đó \mathbf{I} là ma trận đơn vị bậc m . Nếu một ma trận thỏa mãn điều kiện 1.22, ta gọi nó là *ma trận trực giao* (*orthogonal matrix*). Ma trận loại này không được gọi là *ma trận trực chuẩn*, không có định nghĩa cho ma trận trực chuẩn.

Nếu một ma trận vuông phức \mathbf{U} thỏa mãn $\mathbf{U}\mathbf{U}^H = \mathbf{U}^H\mathbf{U} = \mathbf{I}$, ta nói rằng \mathbf{U} là một ma trận *unitary* (*unitary matrix*).

1.9.2 Tính chất của ma trận trực giao

1. $\mathbf{U}^{-1} = \mathbf{U}^T$: nghịch đảo của một ma trận trực giao chính là chuyển vị của nó.
2. Nếu \mathbf{U} là ma trận trực giao thì chuyển vị của nó \mathbf{U}^T cũng là một ma trận trực giao.
3. Định thức của ma trận trực giao bằng 1 hoặc -1 . Điều này có thể suy ra từ việc $\det(\mathbf{U}) = \det(\mathbf{U}^T)$ và $\det(\mathbf{U})\det(\mathbf{U}^T) = \det(\mathbf{I}) = 1$.
4. Ma trận trực giao thể hiện cho phép xoay một vector (xem thêm mục 1.10). Giả sử có hai vector $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ và một ma trận trực giao $\mathbf{U} \in \mathbb{R}^{m \times m}$. Dùng ma trận này để xoay hai vector trên ta được $\mathbf{U}\mathbf{x}, \mathbf{U}\mathbf{y}$. Tích vô hướng của hai vector mới là:

$$(\mathbf{U}\mathbf{x})^T(\mathbf{U}\mathbf{y}) = \mathbf{x}^T\mathbf{U}^T\mathbf{U}\mathbf{y} = \mathbf{x}^T\mathbf{y} \quad (1.23)$$

như vậy phép xoay không làm thay đổi tích vô hướng giữa hai vector.

5. Giả sử $\hat{\mathbf{U}} \in \mathbb{R}^{m \times r}$, $r < m$ là một ma trận con của ma trận trực giao \mathbf{U} được tạo bởi r cột của \mathbf{U} , ta sẽ có $\hat{\mathbf{U}}^T\hat{\mathbf{U}} = \mathbf{I}_r$. Việc này có thể được suy ra từ (1.21).

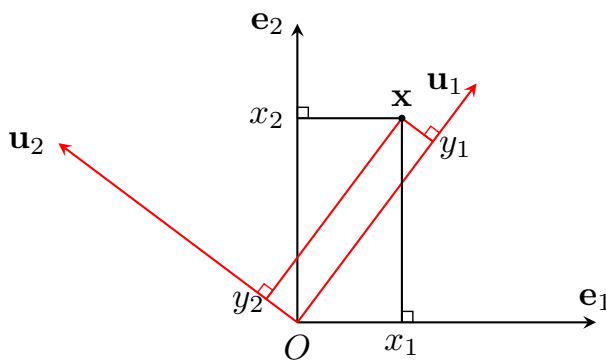
1.10 Biểu diễn vector trong các hệ cơ sở khác nhau

Trong không gian m chiều, tọa độ của mỗi điểm được xác định dựa trên một hệ tọa độ nào đó. Ở các hệ tọa độ khác nhau, hiển nhiên là tọa độ của mỗi điểm cũng khác nhau.

Tập hợp các vector $\mathbf{e}_1, \dots, \mathbf{e}_m$ mà mỗi vector \mathbf{e}_i có đúng 1 phần tử khác 0 ở thành phần thứ i và phần tử đó bằng 1, được gọi là hệ cơ sở đơn vị (hoặc hệ đơn vị, hoặc hệ chính tắc) trong không gian m chiều. Nếu xếp các vector \mathbf{e}_i , $i = 1, 2, \dots, m$ theo đúng thứ tự đó, ta sẽ được ma trận đơn vị m chiều.

Mỗi vector cột $\mathbf{x} = [x_1, x_2, \dots, x_m] \in \mathbb{R}^m$ có thể coi là một tổ hợp tuyến tính của các vector trong hệ cơ sở chính tắc:

$$\mathbf{x} = x_1\mathbf{e}_1 + x_2\mathbf{e}_2 + \dots + x_m\mathbf{e}_m \quad (1.24)$$



Hình 1.1: Chuyển đổi tọa độ trong các hệ cơ sở khác nhau. Trong hệ tọa độ Oe_1e_2 , \mathbf{x} có tọa độ là (x_1, x_2) . Trong hệ tọa độ Ou_1u_2 , \mathbf{x} có tọa độ là (y_1, y_2) .

Giả sử có một hệ cơ sở khác $\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_m$ (các vector này độc lập tuyến tính), biểu diễn của vector \mathbf{x} trong hệ cơ sở mới này có dạng:

$$\mathbf{x} = y_1\mathbf{u}_1 + y_2\mathbf{u}_2 + \dots + y_m\mathbf{u}_m = \mathbf{U}\mathbf{y} \tag{1.25}$$

với $\mathbf{U} = [\mathbf{u}_1 \dots \mathbf{u}_m]$. Lúc này, vector $\mathbf{y} = [y_1, y_2, \dots, y_m]^T$ chính là biểu diễn của \mathbf{x} trong hệ cơ sở mới. Biểu diễn này là duy nhất vì $\mathbf{y} = \mathbf{U}^{-1}\mathbf{x}$.

Trong các ma trận đóng vai trò như hệ cơ sở, các ma trận trực giao, tức $\mathbf{U}^T\mathbf{U} = \mathbf{I}$, được quan tâm nhiều hơn vì nghịch đảo của chúng chính là chuyển vị của chúng: $\mathbf{U}^{-1} = \mathbf{U}^T$. Khi đó, \mathbf{y} có thể được tính một cách nhanh chóng $\mathbf{y} = \mathbf{U}^T\mathbf{x}$. Từ đó suy ra: $y_i = \mathbf{x}^T\mathbf{u}_i = \mathbf{u}_i^T\mathbf{x}, i = 1, \dots, m$. Dưới góc nhìn hình học, hệ trực giao tạo thành một hệ trục tọa độ Descartes vuông góc mà chúng ta đã quen thuộc trong không gian hai chiều hoặc ba chiều.

Có thể nhận thấy rằng vector $\mathbf{0}$ được biểu diễn như nhau trong mọi hệ cơ sở. Hình 1.1 là một ví dụ về việc chuyển hệ cơ sở trong không gian hai chiều.

Việc chuyển đổi hệ cơ sở sử dụng ma trận trực giao có thể được coi như một phép xoay trục tọa độ. Nhìn theo một cách khác, đây cũng chính là một phép xoay vector dữ liệu theo chiều ngược lại, nếu ta coi các trục tọa độ là cố định. Trong chương Principle Component Analysis, chúng ta sẽ thấy được một ứng dụng quan trọng của việc đổi hệ cơ sở.

1.11 Trị riêng và vector riêng

1.11.1 Định nghĩa

Cho một ma trận vuông $\mathbf{A} \in \mathbb{R}^{n \times n}$, một vector $\mathbf{x} \in \mathbb{R}^n (\mathbf{x} \neq \mathbf{0})$ và một số vô hướng (có thể thực hoặc phức) λ . Nếu $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, thì ta nói λ và \mathbf{x} là một cặp *trị riêng, vector riêng* (*eigenvalue, eigenvector*) của ma trận \mathbf{A} .

Từ định nghĩa ta cũng có $(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = \mathbf{0}$, tức \mathbf{x} nằm trong null space của $\mathbf{A} - \lambda\mathbf{I}$. Vì $\mathbf{x} \neq \mathbf{0}$, $\mathbf{A} - \lambda\mathbf{I}$ là một ma trận không khả nghịch. Nói cách khác $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$, tức λ là nghiệm của phương trình $\det(\mathbf{A} - t\mathbf{I}) = 0$. Định thức này là một đa thức bậc n của t , được gọi là *đa thức đặc trưng* (*characteristic polynomial*) của \mathbf{A} , được ký hiệu là $p_{\mathbf{A}}(t)$. Tập hợp tất cả các trị riêng của một ma trận vuông còn được gọi là *phổ* (*spectrum*) của ma trận đó.

1.11.2 Tính chất

1. Nếu \mathbf{x} là một vector riêng của \mathbf{A} ứng với λ thì $k\mathbf{x}$, $\forall k \neq 0$ cũng là vector riêng ứng với trị riêng đó. Nếu $\mathbf{x}_1, \mathbf{x}_2$ là hai vector riêng ứng với cùng trị riêng λ , thì tổng của chúng cũng là một vector ứng với trị riêng đó. Từ đó suy ra tập hợp các vector riêng ứng với một trị riêng của một ma trận vuông tạo thành một không gian vector con, thường được gọi là *không gian riêng* (*eigenspace*) ứng với trị riêng đó.
2. Mọi ma trận vuông bậc n đều có n trị riêng (kể cả lặp) và có thể là các số phức.
3. Tích của tất cả các trị riêng của một ma trận bằng định thức của ma trận đó. Tổng tất cả các trị riêng của một ma trận bằng tổng các phần tử trên đường chéo của ma trận đó.
4. Phổ của một ma trận bằng phổ của ma trận chuyển vị của nó.
5. Nếu \mathbf{A}, \mathbf{B} là các ma trận vuông cùng bậc thì $p_{\mathbf{AB}}(t) = p_{\mathbf{BA}}(t)$. Điều này nghĩa là, mặc dù tích của hai ma trận không có tính chất giao hoán, đa thức đặc trưng của \mathbf{AB} và \mathbf{BA} là như nhau. Tức phổ của hai tích này là trùng nhau.
6. Với ma trận đối xứng (hoặc tổng quát, Hermitian), tất cả các trị riêng của nó đều là các số thực. Thật vậy, giả sử λ là một trị riêng của một ma trận Hermitian \mathbf{A} và \mathbf{x} là một vector riêng ứng với trị riêng đó. Từ định nghĩa ta suy ra:

$$\mathbf{Ax} = \lambda\mathbf{x} \Rightarrow (\mathbf{Ax})^H = \bar{\lambda}\mathbf{x}^H \Rightarrow \bar{\lambda}\mathbf{x}^H = \mathbf{x}^H\mathbf{A} \quad (1.26)$$

với $\bar{\lambda}$ là liên hiệp phức của số vô hướng λ . Nhân cả hai vế vào bên phải với \mathbf{x} ta có:

$$\bar{\lambda}\mathbf{x}^H\mathbf{x} = \mathbf{x}^H\mathbf{Ax} = \lambda\mathbf{x}^H\mathbf{x} \Rightarrow (\lambda - \bar{\lambda})\mathbf{x}^H\mathbf{x} = 0 \quad (1.27)$$

vì $\mathbf{x} \neq 0$ nên $\mathbf{x}^H\mathbf{x} \neq 0$. Từ đó suy ra $\bar{\lambda} = \lambda$, tức λ phải là một số thực.

7. Nếu (λ, \mathbf{x}) là một cặp trị riêng, vector riêng của một ma trận khả nghịch \mathbf{A} , thì $(\frac{1}{\lambda}, \mathbf{x})$ là một cặp trị riêng, vector riêng của \mathbf{A}^{-1} , vì $\mathbf{Ax} = \lambda\mathbf{x} \Rightarrow \frac{1}{\lambda}\mathbf{x} = \mathbf{A}^{-1}\mathbf{x}$.

1.12 Chéo hoá ma trận

Việc phân tích một đại lượng toán học ra thành các đại lượng nhỏ hơn mang lại nhiều hiệu quả. Phân tích một số thành tích các thừa số nguyên tố giúp kiểm tra một số có bao nhiêu ước số. Phân tích đa thức thành nhân tử giúp tìm nghiệm của đa thức. Việc phân tích một ma trận thành tích của các ma trận có dạng đặc biệt khác (quá trình này được gọi là *matrix decomposition*) cũng mang lại nhiều lợi ích trong việc giải hệ phương trình một cách hiệu quả, tính lũy thừa của ma trận, xấp xỉ ma trận, nén dữ liệu, phân cụm dữ liệu, v.v. Trong mục này, chúng ta sẽ ôn lại một phương pháp matrix decomposition quen thuộc—phương pháp chéo hoá ma trận (*diagonalization* hoặc *eigendecomposition*).

Giả sử $\mathbf{x}_1, \dots, \mathbf{x}_n \neq \mathbf{0}$ là các vector riêng của một ma trận vuông \mathbf{A} ứng với các trị riêng $\lambda_1, \dots, \lambda_n$ (có thể lặp hoặc là các số phức) của nó. Tức là $\mathbf{Ax}_i = \lambda_i\mathbf{x}_i$, $\forall i = 1, \dots, n$.

Đặt $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, và $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$, ta sẽ có $\mathbf{A}\mathbf{X} = \mathbf{X}\mathbf{\Lambda}$. Hơn nữa, nếu các trị riêng $\lambda_1, \dots, \lambda_n$ là độc lập tuyến tính, ma trận \mathbf{X} là một ma trận khả nghịch. Khi đó ta có thể viết \mathbf{A} dưới dạng tích của ba ma trận:

$$\mathbf{A} = \mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1} \quad (1.28)$$

Các vector riêng \mathbf{x}_i thường được chọn sao cho $\mathbf{x}_i^T \mathbf{x}_i = 1$. Cách biểu diễn một ma trận như (1.28) được gọi là *eigendecomposition* vì nó tách ra thành tích của các ma trận đặc biệt dựa trên vector riêng (eigenvectors) và trị riêng (eigenvalues). Ma trận các trị riêng $\mathbf{\Lambda}$ là một ma trận đường chéo. Vì vậy, cách khai triển này cũng có tên gọi là chéo hoá ma trận.

Tính chất:

1. Khái niệm chéo hoá ma trận chỉ áp dụng với ma trận vuông. Vì không có định nghĩa vector riêng hay trị riêng cho ma trận không vuông.
2. Không phải ma trận vuông nào cũng có thể chéo hoá được (*diagonalizable*). Một ma trận vuông bậc n là chéo hoá được nếu và chỉ nếu nó có đủ n trị riêng độc lập tuyến tính.
3. Nếu một ma trận là chéo hoá được, có nhiều hơn một cách chéo hoá ma trận đó. Chỉ cần đổi vị trí của các λ_i và vị trí tương ứng các cột của \mathbf{X} , ta sẽ có một cách chéo hoá mới.
4. Nếu \mathbf{A} có thể viết được dưới dạng (1.28), khi đó các lũy thừa có nó cũng chéo hoá được. Cụ thể:

$$\mathbf{A}^2 = (\mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1})(\mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1}) = \mathbf{X}\mathbf{\Lambda}^2\mathbf{X}^{-1}; \quad \mathbf{A}^k = \mathbf{X}\mathbf{\Lambda}^k\mathbf{X}^{-1}, \quad \forall k \in \mathbb{N} \quad (1.29)$$

Xin chú ý rằng nếu λ và \mathbf{x} là một cặp (trị riêng, vector riêng) của \mathbf{A} , thì λ^k và \mathbf{x} là một cặp (trị riêng, vector riêng) của \mathbf{A}^k . Thật vậy, $\mathbf{A}^k \mathbf{x} = \mathbf{A}^{k-1}(\mathbf{A}\mathbf{x}) = \lambda \mathbf{A}^{k-1} \mathbf{x} = \dots = \lambda^k \mathbf{x}$.

5. Nếu \mathbf{A} khả nghịch, thì $\mathbf{A}^{-1} = (\mathbf{X}\mathbf{\Lambda}\mathbf{X}^{-1})^{-1} = \mathbf{X}\mathbf{\Lambda}^{-1}\mathbf{X}^{-1}$. Vậy chéo hoá ma trận cũng có ích trong việc tính ma trận nghịch đảo.

1.13 Ma trận xác định dương

1.13.1 Định nghĩa

Một ma trận đối xứng² $\mathbf{A} \in \mathbb{R}^{n \times n}$ được gọi là *xác định dương* (*positive definite*) nếu:

$$\mathbf{x}^T \mathbf{A} \mathbf{x} > 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq \mathbf{0}. \quad (1.30)$$

Một ma trận đối xứng $\mathbf{A} \in \mathbb{R}^{n \times n}$ được gọi là *nửa xác định dương* (*positive semidefinite*) nếu:

$$\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq \mathbf{0}. \quad (1.31)$$

Trên thực tế, ma trận nửa xác định dương, được viết tắt là *PSD*, được sử dụng nhiều hơn.

² Chú ý, tồn tại những ma trận không đối xứng thoả mãn điều kiện (1.30). Ta sẽ không xét những ma trận này.

Ma trận *xác định âm* (*negative definite*) và *nửa xác định âm* (*negative semi-definite*) cũng được định nghĩa tương tự.

Ký hiệu $\mathbf{A} \succ 0, \succeq 0, \prec 0, \preceq 0$ được dùng để chỉ một ma trận là xác định dương, nửa xác định dương, xác định âm, nửa xác định âm, theo thứ tự đó. Ký hiệu $\mathbf{A} \succ \mathbf{B}$ cũng được dùng để chỉ ra rằng $\mathbf{A} - \mathbf{B} \succ 0$.

Mở rộng, một ma trận phức, Hermitian $\mathbf{A} \in \mathbb{C}^{n \times n}$ được gọi là xác định dương nếu:

$$\mathbf{x}^H \mathbf{A} \mathbf{x} > 0, \forall \mathbf{x} \in \mathbb{C}^n, \mathbf{x} \neq \mathbf{0}. \quad (1.32)$$

Ví dụ, $\mathbf{A} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$ là nửa xác định dương vì với mọi vector $\mathbf{x} = \begin{bmatrix} u \\ v \end{bmatrix}$, ta có:

$$\mathbf{x}^T \mathbf{A} \mathbf{x} = [u \ v] \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = u^2 + v^2 - 2uv = (u - v)^2 \geq 0, \forall u, v \in \mathbb{R} \quad (1.33)$$

1.13.2 Tính chất

1. Mọi trị riêng của một ma trận xác định dương đều là một số thực dương.

Trước hết, các trị riêng của các ma trận dạng này là số thực vì các ma trận đều là đối xứng. Để chứng minh chúng là các số thực dương, ta giả sử λ là một trị riêng của một ma trận xác định dương \mathbf{A} và $\mathbf{x} \neq \mathbf{0}$ là một vector riêng ứng với trị riêng đó. Nhân vào bên trái cả hai vế của $\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$ với \mathbf{x}^H ta có:

$$\lambda \mathbf{x}^H \mathbf{x} = \mathbf{x}^H \mathbf{A} \mathbf{x} > 0 \quad (1.34)$$

(ở đây Hermitian được dùng để xét tổng quát cho cả trường hợp ma trận phức). Vì $\mathbf{x}^H \mathbf{x}$ luôn dương với mọi \mathbf{x} nên ta phải có $\lambda > 0$. Tương tự, ta có thể chứng minh được rằng mọi trị riêng của một ma trận nửa xác định dương là không âm.

2. Mọi ma trận xác định dương là khả nghịch. Hơn nữa, định thức của nó là một số dương.

Điều này được trực tiếp suy ra từ tính chất 1. Nhắc lại rằng định thức của một ma trận bằng tích tất cả các trị riêng của nó.

3. Tiêu chuẩn Sylvester: Một ma trận Hermitian là xác định dương nếu và chỉ nếu mọi *leading principal minors* của nó là dương. Một ma trận Hermitian là nửa xác định dương nếu mọi *principal minors* của nó là không âm. Đây là một tiêu chuẩn để kiểm tra một ma trận Hermitian $\mathbf{A} \in \mathbb{R}^n$ có là (nửa) xác định dương hay không. Ở đây, *leading principal minors* và *principal minors* được định nghĩa như sau:

Gọi \mathcal{I} là một tập con bất kỳ của $\{1, 2, \dots, n\}$, $\mathbf{A}_{\mathcal{I}}$ là ma trận con của \mathbf{A} nhận được bằng cách trích ra các hàng và cột có chỉ số nằm trong \mathcal{I} của \mathbf{A} . Khi đó, $\mathbf{A}_{\mathcal{I}}$ và $\det(\mathbf{A}_{\mathcal{I}})$ lần lượt được gọi là một *ma trận con chính* (*principal submatrix*) và *principal minor* của \mathbf{A} . Nếu \mathcal{I} chỉ bao gồm các số tự nhiên liên tiếp từ 1 đến $k \leq n$, ta nói $\mathbf{A}_{\mathcal{I}}$ và $\det(\mathbf{A}_{\mathcal{I}})$ lần lượt là một *leading principal submatrix* và *leading principal minor* bậc k của \mathbf{A} .

4. $\mathbf{A} = \mathbf{B}^H \mathbf{B}$ là nửa xác định dương với mọi ma trận \mathbf{B} (\mathbf{B} không nhất thiết vuông).

Thật vậy, với mọi vector $\mathbf{x} \neq \mathbf{0}$ với chiều phù hợp, $\mathbf{x}^H \mathbf{A} \mathbf{x} = \mathbf{x}^H \mathbf{B}^H \mathbf{B} \mathbf{x} = (\mathbf{B} \mathbf{x})^H (\mathbf{B} \mathbf{x}) \geq 0$.

5. Khai triển Cholesky (Cholesky decomposition): Mọi ma trận Hermitian, nửa xác định dương \mathbf{A} đều biểu diễn được duy nhất dưới dạng $\mathbf{A} = \mathbf{L} \mathbf{L}^H$, trong đó \mathbf{L} là một ma trận tam giác dưới với các thành phần trên đường chéo là thực dương.

6. Nếu \mathbf{A} là một ma trận nửa xác định dương, thì $\mathbf{x}^T \mathbf{A} \mathbf{x} = 0 \Leftrightarrow \mathbf{A} \mathbf{x} = \mathbf{0}$.

Nếu $\mathbf{A} \mathbf{x} = \mathbf{0} \Rightarrow \mathbf{x}^T \mathbf{A} \mathbf{x} = 0$ một cách hiển nhiên.

Nếu $\mathbf{x}^T \mathbf{A} \mathbf{x} = 0$. Với vector $\mathbf{y} \neq \mathbf{0}$ bất kỳ có cùng kích thước với \mathbf{x} , xét hàm số sau đây:

$$f(\lambda) = (\mathbf{x} + \lambda \mathbf{y})^T \mathbf{A} (\mathbf{x} + \lambda \mathbf{y}) \quad (1.35)$$

Hàm số này không âm với mọi λ vì \mathbf{A} là một ma trận nửa xác định dương. Đây là một tam thức bậc hai của λ :

$$f(\lambda) = \mathbf{y}^T \mathbf{A} \mathbf{y} \lambda^2 + 2\mathbf{y}^T \mathbf{A} \mathbf{x} \lambda + \mathbf{x}^T \mathbf{A} \mathbf{x} = \mathbf{y}^T \mathbf{A} \mathbf{y} \lambda^2 + 2\mathbf{y}^T \mathbf{A} \mathbf{x} \lambda \quad (1.36)$$

Xét hai trường hợp:

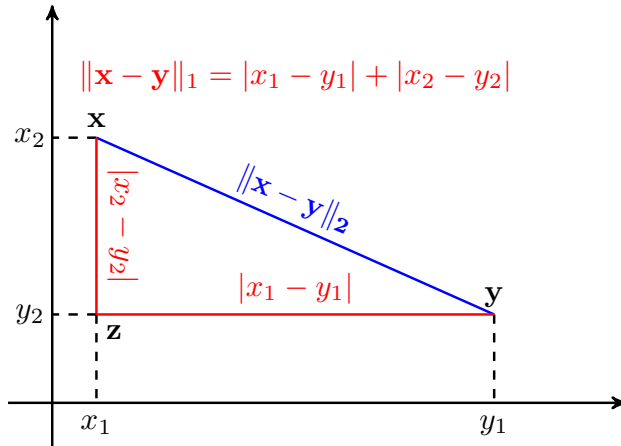
- $\mathbf{y}^T \mathbf{A} \mathbf{y} = 0$. Khi đó, $f(\lambda) = 2\mathbf{y}^T \mathbf{A} \mathbf{x} \lambda \geq 0, \forall \lambda$ nếu và chỉ nếu $\mathbf{y}^T \mathbf{A} \mathbf{x} = 0$.
- $\mathbf{y}^T \mathbf{A} \mathbf{y} > 0$. Khi đó tam thức bậc hai $f(\lambda) \geq 0, \forall \lambda$ nếu và chỉ nếu $\Delta' = (\mathbf{y}^T \mathbf{A} \mathbf{x})^2 \leq 0$ vì hệ số ứng với thành phần bậc hai bằng $\mathbf{y}^T \mathbf{A} \mathbf{y} > 0$. Điều này cũng đồng nghĩa với việc $\mathbf{y}^T \mathbf{A} \mathbf{x} = 0$

Tóm lại, $\mathbf{y}^T \mathbf{A} \mathbf{x} = 0, \forall \mathbf{y} \neq \mathbf{0}$. Điều này chỉ xảy ra nếu $\mathbf{A} \mathbf{x} = \mathbf{0}$. □

1.14 Chuẩn của vector và ma trận

Trong không gian một chiều, khoảng cách giữa hai điểm là 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 đại lượng 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.

Việc đo khoảng cách giữa hai điểm dữ liệu nhiều chiều, tức hai vector, là rất cần thiết trong Machine Learning. Và đó chính là lý do mà khái niệm *chuẩn (norm)* ra đời. Để xác định khoảng cách giữa hai vector \mathbf{y} và \mathbf{z} , người ta thường áp dụng một hàm số lên vector hiệu $\mathbf{x} = \mathbf{y} - \mathbf{z}$. Hàm số này cần có một vài tính chất đặc biệt.



Hình 1.2: Minh họa ℓ_1 norm và ℓ_2 norm trong không gian hai chiều. ℓ_2 norm chính là khoảng cách giữa hai điểm trong mặt phẳng. Trong khi đó ℓ_1 norm là quãng đường ngắn nhất giữa hai điểm nếu chỉ được đi theo các đường song song với các trục tọa độ.

Định nghĩa 1.1: Norm

Một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ được gọi là một norm nếu nó thỏa mãn ba điều kiện sau đây:

1. $f(\mathbf{x}) \geq 0$. Dấu 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} 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 $\mathbf{x}_1 = \mathbf{y} - \mathbf{w}$, $\mathbf{x}_2 = \mathbf{w} - \mathbf{z}$ với \mathbf{w} là một điểm bất kỳ trong cùng không gian.

1.14.1 Một số chuẩn vector thường dùng

Độ dài Euclid của một vector $\mathbf{x} \in \mathbb{R}^n$ chính là một norm, norm này được gọi là ℓ_2 norm hoặc Euclidean norm:

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \quad (1.37)$$

Bình phương của ℓ_2 norm chính là tích vô hướng của một vector với chính nó, $\|\mathbf{x}\|_2^2 = \mathbf{x}^T \mathbf{x}$. Với p là một số không nhỏ hơn 1 bất kỳ, hàm số:

$$\|\mathbf{x}\|_p = (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{\frac{1}{p}} \quad (1.38)$$

được chứng minh thỏa mãn ba điều kiện của norm, và được gọi là ℓ_p norm.

Có một vài giá trị của p thường được dùng:

1. Khi $p = 2$ chúng ta có ℓ_2 norm như ở trên.

2. Khi $p = 1$ chúng ta có ℓ_1 norm: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots + |x_n|$ là tổng các trị tuyệt đối của từng phần tử của \mathbf{x} . Hình 1.2 là một ví dụ sánh ℓ_1 norm và ℓ_2 norm trong không gian hai chiều. Norm 2 (màu xanh) chính là đường thẳng *chim bay* nối giữa hai vector \mathbf{x} và \mathbf{y} . Khoảng cách ℓ_1 norm 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 như đường chim bay.
3. Khi $p \rightarrow \infty$, giả sử $i = \arg \max_{j=1,2,\dots,n} |x_j|$. Khi đó:

$$\|\mathbf{x}\|_p = |x_i| \left(1 + \left| \frac{x_1}{x_i} \right|^p + \dots + \left| \frac{x_{i-1}}{x_i} \right|^p + \left| \frac{x_{i+1}}{x_i} \right|^p + \dots + \left| \frac{x_n}{x_i} \right|^p \right)^{\frac{1}{p}} \quad (1.39)$$

Ta thấy rằng:

$$\lim_{p \rightarrow \infty} \left(1 + \left| \frac{x_1}{x_i} \right|^p + \dots + \left| \frac{x_{i-1}}{x_i} \right|^p + \left| \frac{x_{i+1}}{x_i} \right|^p + \dots + \left| \frac{x_n}{x_i} \right|^p \right)^{\frac{1}{p}} = 1 \quad (1.40)$$

vì đại lượng trong dấu ngoặc đơn không vượt quá n , ta sẽ có:

$$\|\mathbf{x}\|_\infty \triangleq \lim_{p \rightarrow \infty} \|\mathbf{x}\|_p = |x_i| = \max_{j=1,2,\dots,n} |x_j| \quad (1.41)$$

1.14.2 Chuẩn Frobenius 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.

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$$

Chú ý rằng ℓ_2 norm $\|\mathbf{A}\|_2$ là một norm khác của ma trận, không phổ biến bằng Frobenius norm. Bạn đọc có thể xem ℓ_2 norm của ma trận trong Phụ lục A.

1.14.3 Vết của ma trận

Vết (trace) của một ma trận vuông là tổng tất cả các phần tử trên đường chéo chính của nó.

Vết của một ma trận được \mathbf{A} được ký hiệu là $\text{trace}(\mathbf{A})$. Hàm số trace xác định trên tập các ma trận vuông được sử dụng rất nhiều trong tối ưu vì những tính chất đẹp của nó.

Các tính chất quan trọng của hàm trace , với giả sử rằng các ma trận trong hàm trace là vuông và các phép nhân ma trận thực hiện được:

- Một ma trận vuông bất kỳ và chuyển vị của nó có *trace* bằng nhau $\text{trace}(\mathbf{A}) = \text{trace}(\mathbf{A}^T)$. Việc này khá hiển nhiên vì phép chuyển vị không làm thay đổi các phần tử trên đường chéo chính của một ma trận.

- *trace của một tổng bằng tổng các trace*: $\text{trace}(\sum_{i=1}^k \mathbf{A}_i) = \sum_{i=1}^k \text{trace}(\mathbf{A}_i)$.
- $\text{trace}(k\mathbf{A}) = k\text{trace}(\mathbf{A})$ với k là một số vô hướng bất kỳ.
- $\text{trace}(\mathbf{A}) = \sum_{i=1}^D \lambda_i$ với \mathbf{A} là một ma trận vuông và $\lambda_i, i = 1, 2, \dots, N$ là toàn bộ các trị riêng của nó, có thể phức hoặc lặp. Việc chứng minh tính chất này có thể được dựa trên ma trận đặc trưng của \mathbf{A} và định lý Viète.
- $\text{trace}(\mathbf{AB}) = \text{trace}(\mathbf{BA})$. Đẳng thức này được suy ra từ việc đa thức đặc trưng của \mathbf{AB} và \mathbf{BA} là như nhau. Bạn đọc cũng có thể chứng minh bằng cách tính trực tiếp các phần tử trên đường chéo chính của \mathbf{AB} và \mathbf{BA} .
- $\text{trace}(\mathbf{ABC}) = \text{trace}(\mathbf{BCA})$ nhưng $\text{trace}(\mathbf{ABC})$ không đồng nhất với $\text{trace}(\mathbf{ACB})$.
- Nếu \mathbf{X} là một ma trận khả nghịch cùng chiều với \mathbf{A} :

$$\text{trace}(\mathbf{XAX}^{-1}) = \text{trace}(\mathbf{X}^{-1}\mathbf{XA}) = \text{trace}(\mathbf{A}) \quad (1.42)$$

- $\|\mathbf{A}\|_F^2 = \text{trace}(\mathbf{A}^T \mathbf{A}) = \text{trace}(\mathbf{AA}^T)$ với \mathbf{A} là một ma trận bất kỳ. Từ đây ta cũng suy ra $\text{trace}(\mathbf{AA}^T) \geq 0$ với mọi ma trận \mathbf{A} .

Giải tích ma trận

Trong chương này, nếu không nói gì thêm, chúng ta giả sử rằng các đạo hàm tồn tại. Tài liệu tham khảo chính của chương là *Matrix calculus–Stanford* (<https://goo.gl/BjTPLr>).

2.1 Đạo hàm của hàm trả về một số vô hướng

Đạo hàm bậc nhất (*first-order gradient*) hay viết gọn là đạo hàm (*gradient*) của một hàm số $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ theo \mathbf{x} được định nghĩa là

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \quad (2.1)$$

trong đó $\frac{\partial f(\mathbf{x})}{\partial x_i}$ là đạo hàm riêng (*partial derivative*) của hàm số theo thành phần thứ i của vector \mathbf{x} . Đạo hàm này được lấy khi tất cả các biến, ngoài x_i , được giả sử là hằng số. Nếu không có thêm biến nào khác, $\nabla_{\mathbf{x}} f(\mathbf{x})$ thường được viết gọn là $\nabla f(\mathbf{x})$. **Đạo hàm của hàm số này là một vector có cùng chiều với vector đang được lấy đạo hàm.** Tức nếu vector được viết ở dạng cột thì đạo hàm cũng phải được 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, với $\mathbb{S}^n \in \mathbb{R}^{n \times n}$ là tập các ma trận vuông đối xứng bậc n .

$$\nabla^2 f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix} \in \mathbb{S}^n \quad (2.2)$$

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

$$\nabla f(\mathbf{X}) = \begin{bmatrix} \frac{\partial f(\mathbf{X})}{\partial x_{11}} & \frac{\partial f(\mathbf{X})}{\partial x_{12}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{1m}} \\ \frac{\partial f(\mathbf{X})}{\partial x_{21}} & \frac{\partial f(\mathbf{X})}{\partial x_{22}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{2m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(\mathbf{X})}{\partial x_{n1}} & \frac{\partial f(\mathbf{X})}{\partial x_{n2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{nm}} \end{bmatrix} \in \mathbb{R}^{n \times m} \quad (2.3)$$

Chiều của đạo hàm

Đạo hàm của hàm số $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ là một ma trận trong $\mathbb{R}^{m \times n}$, $\forall m, n \in \mathbb{N}^*$.

Cụ thể, để tính đạo hàm của một hàm $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$, ta tính đạo hàm riêng 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 sắp xếp các đạo hàm riêng tính được theo đúng thứ tự trong 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 $\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 \partial x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 2 - \sin(x_1) & 2 \\ 2 & 0 \end{bmatrix}$

Chú ý rằng *Hessian* luôn là một ma trận đối xứng.

2.2 Đạo hàm của hàm trả về một vector

Những hàm số trả về một vector, hoặc gọn hơn hàm trả về vector được gọi là *vector-valued function* trong tiếng Anh.

Xét một hàm trả về vector với đầu vào là một số thực $v(x) : \mathbb{R} \rightarrow \mathbb{R}^n$:

$$v(x) = \begin{bmatrix} v_1(x) \\ v_2(x) \\ \vdots \\ v_n(x) \end{bmatrix} \quad (2.4)$$

Đạo hàm của hàm số này theo x là một **vector hàng** như sau:

$$\nabla v(x) \triangleq \left[\frac{\partial v_1(x)}{\partial x} \quad \frac{\partial v_2(x)}{\partial x} \quad \dots \quad \frac{\partial v_n(x)}{\partial x} \right] \quad (2.5)$$

Đạo hàm bậc hai của hàm số này có dạng

$$\nabla^2 v(x) \triangleq \left[\frac{\partial^2 v_1(x)}{\partial x^2} \quad \frac{\partial^2 v_2(x)}{\partial x^2} \quad \dots \quad \frac{\partial^2 v_n(x)}{\partial x^2} \right] \quad (2.6)$$

Ví dụ: Cho một vector $\mathbf{a} \in \mathbb{R}^n$ và một hàm số *vector-valued* $v(x) = \mathbf{x}\mathbf{a}$, đạo hàm bậc nhất và Hessian của nó lần lượt là

$$\nabla v(x) = \mathbf{a}^T, \quad \nabla^2 v(x) = \mathbf{0} \in \mathbb{R}^{1 \times n} \quad (2.7)$$

Xét một hàm trả về vector với **đầu vào là một vector** $h(\mathbf{x}) : \mathbb{R}^k \rightarrow \mathbb{R}^n$, đạo hàm bậc nhất của nó là

$$\nabla h(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial h_1(\mathbf{x})}{\partial x_1} & \frac{\partial h_2(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_1} \\ \frac{\partial h_1(\mathbf{x})}{\partial x_2} & \frac{\partial h_2(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_1(\mathbf{x})}{\partial x_k} & \frac{\partial h_2(\mathbf{x})}{\partial x_k} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_k} \end{bmatrix} = \left[\nabla h_1(\mathbf{x}) \quad \nabla h_2(\mathbf{x}) \quad \dots \quad \nabla h_n(\mathbf{x}) \right] \in \mathbb{R}^{k \times n} \quad (2.8)$$

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 *mảng ba chiều*, chúng ta sẽ không nhắc đến ở đây.

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.

2.3 Tính chất quan trọng của đạo hàm

2.3.1 Quy tắc tích (Product rule)

Để cho tổng quát, ta giả sử biến đầu vào là một ma trận. Giả sử rằng các hàm số có chiều phù hợp để các phép nhân thực hiện được. Ta có:

$$\nabla (f(\mathbf{X})^T g(\mathbf{X})) = (\nabla f(\mathbf{X})) g(\mathbf{X}) + (\nabla g(\mathbf{X})) f(\mathbf{X}) \quad (2.9)$$

Biểu thức này giống như biểu thức chúng ta đã quen thuộc:

$$(f(x)g(x))' = f'(x)g(x) + g'(x)f(x)$$

Chú ý rằng với tích của vector và ma trận, ta không được sử dụng tính chất giao hoán.

2.3.2 Quy tắc chuỗi (Chain rule)

Khi có các hàm hợp thì

$$\nabla_{\mathbf{x}}g(f(\mathbf{X})) = (\nabla_{\mathbf{x}}f)^T(\nabla_{\mathbf{f}}g) \quad (2.10)$$

Quy tắc này cũng giống với quy tắc trong hàm một biến:

$$(g(f(x)))' = f'(x)g'(f)$$

Một lưu ý nhỏ nhưng quan trọng khi làm việc với tích các ma trận là sự phù hợp về kích thước của các ma trận trong tích.

2.4 Đạo hàm của các hàm số thường gặp

2.4.1 $f(\mathbf{x}) = \mathbf{a}^T \mathbf{x}$

Giả sử $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$, ta viết lại $f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} = a_1x_1 + a_2x_2 + \dots + a_nx_n$

Có thể nhận thấy rằng $\frac{\partial f(\mathbf{x})}{\partial x_i} = a_i, \forall i = 1, 2, \dots, n$.

Vậy, $\nabla(\mathbf{a}^T \mathbf{x}) = [a_1 \ a_2 \ \dots \ a_n]^T = \mathbf{a}$. Ngoài ra, vì $\mathbf{a}^T \mathbf{x} = \mathbf{x}^T \mathbf{a}$ nên $\nabla(\mathbf{x}^T \mathbf{a}) = \mathbf{a}$.

2.4.2 $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$

Đây là một hàm trả về vector $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ 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 (2.8), và công thức đạo hàm của $\mathbf{a}_i\mathbf{x}$, ta có thể suy ra

$$\nabla_{\mathbf{x}}(\mathbf{A}\mathbf{x}) = [\mathbf{a}_1^T \ \mathbf{a}_2^T \ \dots \ \mathbf{a}_m^T] = \mathbf{A}^T \quad (2.11)$$

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, là

$$\nabla_{\mathbf{x}} = \mathbf{I}$$

2.4.3 $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$

với $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{n \times n}$. Áp dụng quy tắc tích (2.9) ta có

$$\begin{aligned} \nabla f(\mathbf{x}) &= \nabla ((\mathbf{x}^T) (\mathbf{A} \mathbf{x})) \\ &= (\nabla(\mathbf{x}^T)) \mathbf{A} \mathbf{x} + (\nabla(\mathbf{A} \mathbf{x})) \mathbf{x} \\ &= \mathbf{I} \mathbf{A} \mathbf{x} + \mathbf{A}^T \mathbf{x} \\ &= (\mathbf{A} + \mathbf{A}^T) \mathbf{x} \end{aligned} \tag{2.12}$$

Từ (2.12) và (2.11), 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 $\nabla \mathbf{x}^T \mathbf{A} \mathbf{x} = 2\mathbf{A} \mathbf{x}$, $\nabla^2 \mathbf{x}^T \mathbf{A} \mathbf{x} = 2\mathbf{A}$

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

$$\nabla \|\mathbf{x}\|_2^2 = 2\mathbf{x}, \quad \nabla^2 \|\mathbf{x}\|_2^2 = 2\mathbf{I} \tag{2.13}$$

2.4.4 $f(\mathbf{x}) = \|\mathbf{A} \mathbf{x} - \mathbf{b}\|_2^2$

Có hai cách tính đạo hàm của hàm số này:

Cách 1: Trước hết, biến đổi

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{A} \mathbf{x} - \mathbf{b}\|_2^2 = (\mathbf{A} \mathbf{x} - \mathbf{b})^T (\mathbf{A} \mathbf{x} - \mathbf{b}) = (\mathbf{x}^T \mathbf{A}^T - \mathbf{b}^T) (\mathbf{A} \mathbf{x} - \mathbf{b}) \\ &= \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{b}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{b} \end{aligned}$$

Lấy đạo hàm cho từng số hạng rồi cộng lại ta có

$$\nabla \|\mathbf{A} \mathbf{x} - \mathbf{b}\|_2^2 = 2\mathbf{A}^T \mathbf{A} \mathbf{x} - 2\mathbf{A}^T \mathbf{b} = 2\mathbf{A}^T (\mathbf{A} \mathbf{x} - \mathbf{b})$$

Cách 2: Sử dụng $\nabla(\mathbf{A} \mathbf{x} - \mathbf{b}) = \mathbf{A}^T$ và $\nabla \|\mathbf{x}\|_2^2 = 2\mathbf{x}$ và quy tắc chuỗi (2.10), ta cũng sẽ thu được kết quả tương tự.

2.4.5 $f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}$

Bằng cách viết lại $f(\mathbf{x}) = (\mathbf{a}^T \mathbf{x})(\mathbf{x}^T \mathbf{b})$, ta có thể dùng Quy tắc tích (2.9) và có kết quả

$$\nabla(\mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}) = \mathbf{a} \mathbf{x}^T \mathbf{b} + \mathbf{b} \mathbf{a}^T \mathbf{x} = \mathbf{a} \mathbf{b}^T \mathbf{x} + \mathbf{b} \mathbf{a}^T \mathbf{x} = (\mathbf{a} \mathbf{b}^T + \mathbf{b} \mathbf{a}^T) \mathbf{x},$$

ở đây ta đã sử dụng tính chất $\mathbf{y}^T \mathbf{z} = \mathbf{z}^T \mathbf{y}$.

2.4.6 $f(\mathbf{X}) = \text{trace}(\mathbf{A} \mathbf{X})$

Giả sử $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\mathbf{X} = \mathbb{R}^{m \times n}$, và $\mathbf{B} = \mathbf{A} \mathbf{X} \in \mathbb{R}^{n \times n}$. Theo định nghĩa của trace,

$$f(\mathbf{X}) = \text{trace}(\mathbf{A} \mathbf{X}) = \text{trace}(\mathbf{B}) = \sum_{j=1}^n b_{jj} = \sum_{j=1}^n \sum_{i=1}^n a_{ji} x_{ji} \tag{2.14}$$

Từ đây ta thấy rằng $\frac{\partial f(\mathbf{X})}{\partial x_{ij}} = a_{ji}$. Sử dụng định nghĩa (2.3) ta đạt được $\nabla_{\mathbf{X}} \text{trace}(\mathbf{A} \mathbf{X}) = \mathbf{A}^T$.

Bảng 2.1: Bảng các đạo hàm cơ bản.

$f(\mathbf{x})$	$\nabla f(\mathbf{x})$	$f(\mathbf{X})$	$\nabla_{\mathbf{X}} f(\mathbf{X})$
\mathbf{x}	\mathbf{I}	$\text{trace}(\mathbf{X})$	\mathbf{I}
$\mathbf{a}^T \mathbf{x}$	\mathbf{a}	$\text{trace}(\mathbf{A}^T \mathbf{X})$	\mathbf{A}
$\mathbf{x}^T \mathbf{A} \mathbf{x}$	$(\mathbf{A} + \mathbf{A}^T) \mathbf{x}$	$\text{trace}(\mathbf{X}^T \mathbf{A} \mathbf{X})$	$(\mathbf{A} + \mathbf{A}^T) \mathbf{X}$
$\mathbf{x}^T \mathbf{x} = \ \mathbf{x}\ _2^2$	$2\mathbf{x}$	$\text{trace}(\mathbf{X}^T \mathbf{X}) = \ \mathbf{X}\ _F^2$	$2\mathbf{X}$
$\ \mathbf{A} \mathbf{x} - \mathbf{b}\ _2^2$	$2\mathbf{A}^T (\mathbf{A} \mathbf{x} - \mathbf{b})$	$\ \mathbf{A} \mathbf{X} - \mathbf{B}\ _F^2$	$2\mathbf{A}^T (\mathbf{A} \mathbf{X} - \mathbf{B})$
$\mathbf{a}^T \mathbf{x}^T \mathbf{x} \mathbf{b}$	$2\mathbf{a}^T \mathbf{b} \mathbf{x}$	$\mathbf{a}^T \mathbf{X} \mathbf{b}$	$\mathbf{a} \mathbf{b}^T$
$\mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}$	$(\mathbf{a} \mathbf{b}^T + \mathbf{b} \mathbf{a}^T) \mathbf{x}$	$\text{trace}(\mathbf{A}^T \mathbf{X} \mathbf{B})$	$\mathbf{A} \mathbf{B}^T$

2.4.7 $f(\mathbf{X}) = \mathbf{a}^T \mathbf{X} \mathbf{b}$

Giả sử rằng $\mathbf{a} \in \mathbb{R}^m$, $\mathbf{X} \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^n$. Bạn đọc có thể chứng minh được

$$f(\mathbf{X}) = \sum_{i=1}^m \sum_{j=1}^n x_{ij} a_i b_j$$

Từ đó, sử dụng định nghĩa (2.3) ta sẽ có $\nabla_{\mathbf{X}}(\mathbf{a}^T \mathbf{X} \mathbf{b}^T) = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \dots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \dots & a_2 b_n \\ \dots & \dots & \ddots & \dots \\ a_m b_1 & a_m b_2 & \dots & a_m b_n \end{bmatrix} = \mathbf{a} \mathbf{b}^T$.

2.4.8 $f(\mathbf{X}) = \|\mathbf{X}\|_F^2$

Giả sử $\mathbf{X} \in \mathbb{R}^{n \times n}$, bằng cách viết lại $\|\mathbf{X}\|_F^2 = \sum_{i=1}^n \sum_{j=1}^n x_{ij}^2$, ta có thể suy ra $\frac{\partial f}{\partial x_{ij}} = 2x_{ij}$. Và vì vậy, $\nabla \|\mathbf{X}\|_F^2 = 2\mathbf{X}$.

2.4.9 $f(\mathbf{X}) = \text{trace}(\mathbf{X}^T \mathbf{A} \mathbf{X})$

Giả sử rằng $\mathbf{X} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_n] \in \mathbb{R}^{m \times n}$, $\mathbf{A} \in \mathbb{R}^{m \times m}$. Bằng cách khai triển

$$\mathbf{X}^T \mathbf{A} \mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{bmatrix} \mathbf{A} [\mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_n] = \begin{bmatrix} \mathbf{x}_1^T \mathbf{A} \mathbf{x}_1 & \mathbf{x}_1^T \mathbf{A} \mathbf{x}_2 & \dots & \mathbf{x}_1^T \mathbf{A} \mathbf{x}_n \\ \mathbf{x}_2^T \mathbf{A} \mathbf{x}_1 & \mathbf{x}_2^T \mathbf{A} \mathbf{x}_2 & \dots & \mathbf{x}_2^T \mathbf{A} \mathbf{x}_n \\ \dots & \dots & \ddots & \dots \\ \mathbf{x}_n^T \mathbf{A} \mathbf{x}_1 & \mathbf{x}_n^T \mathbf{A} \mathbf{x}_2 & \dots & \mathbf{x}_n^T \mathbf{A} \mathbf{x}_n \end{bmatrix}, \quad (2.15)$$

ta tính được $\text{trace}(\mathbf{X}^T \mathbf{A} \mathbf{X}) = \sum_{i=1}^n \mathbf{x}_i^T \mathbf{A} \mathbf{x}_i$. Nhắc lại rằng $\nabla_{\mathbf{x}_i} \mathbf{x}_i^T \mathbf{A} \mathbf{x}_i = (\mathbf{A} + \mathbf{A}^T) \mathbf{x}_i$, ta có

$$\nabla_{\mathbf{X}} \text{trace}(\mathbf{X}^T \mathbf{A} \mathbf{X}) = (\mathbf{A} + \mathbf{A}^T) [\mathbf{x}_1 \ \mathbf{x}_2 \ \dots \ \mathbf{x}_n] = (\mathbf{A} + \mathbf{A}^T) \mathbf{X} \quad (2.16)$$

Bằng cách thay $\mathbf{A} = \mathbf{I}$, ta cũng thu được $\nabla_{\mathbf{X}} \text{trace}(\mathbf{X}^T \mathbf{X}) = \nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$.

2.4.10 $f(\mathbf{X}) = \|\mathbf{AX} - \mathbf{B}\|_F^2$

Bằng kỹ thuật hoàn toàn tương tự như đã làm trong mục 2.4.4, ta thu được

$$\nabla_{\mathbf{X}} \|\mathbf{AX} - \mathbf{B}\|_F^2 = 2\mathbf{A}^T(\mathbf{AX} - \mathbf{B})$$

2.5 Bảng các đạo hàm thường gặp

Bảng 2.1 bao gồm đạo hàm của các hàm số thường gặp với biến là vector hoặc đạo hàm.

2.6 Kiểm tra đạo hàm

Việc tính đạo hàm của hàm nhiều biến thông thường khá phức tạp và rất dễ mắc lỗi. Trong thực nghiệm, có một cách để kiểm tra liệu đạo hàm tính được có chính xác không. Cách này dựa trên định nghĩa của đạo hàm cho hàm một biến.

2.6.1 Xấp xỉ đạo hàm của hàm một biến

Theo định nghĩa,

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon} \quad (2.17)$$

Một cách thường được sử dụng là lấy một giá trị ε rất nhỏ, ví dụ 10^{-6} , và sử dụng công thức

$$f'(x) \approx \frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon} \quad (2.18)$$

Cách tính này được gọi là *numerical gradient*. Biểu thức (2.18) được sử dụng rộng rãi hơn để tính *numerical gradient*. Có hai cách giải thích cho vấn đề này.

Bảng giải tích

Chúng ta cùng quay lại một chút với khai triển Taylor. Với ε rất nhỏ, ta có hai xấp xỉ sau:

$$f(x + \varepsilon) \approx f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \frac{f^{(3)}(x)}{6}\varepsilon^3 + \dots \quad (2.19)$$

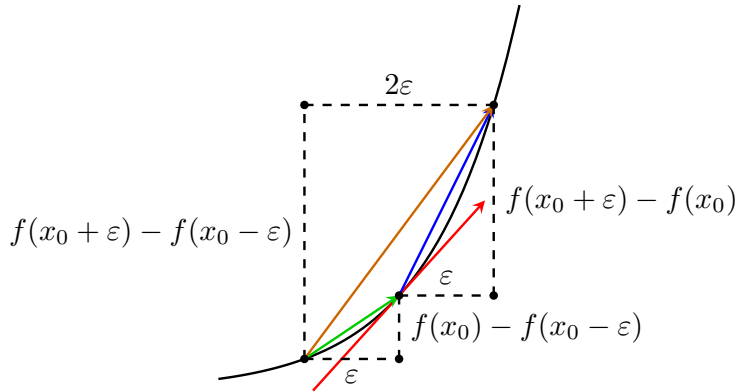
$$f(x - \varepsilon) \approx f(x) - f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 - \frac{f^{(3)}(x)}{6}\varepsilon^3 + \dots \quad (2.20)$$

Từ đó ta có:

$$\frac{f(x + \varepsilon) - f(x)}{\varepsilon} \approx f'(x) + \frac{f''(x)}{2}\varepsilon + \dots = f'(x) + O(\varepsilon) \quad (2.21)$$

$$\frac{f(x + \varepsilon) - f(x - \varepsilon)}{2\varepsilon} \approx f'(x) + \frac{f^{(3)}(x)}{6}\varepsilon^2 + \dots = f'(x) + O(\varepsilon^2) \quad (2.22)$$

trong đó $O()$ là *Big O notation*.



Hình 2.1: Giải thích cách xấp xỉ đạo hàm bằng hình học.

Từ đó, nếu xấp xỉ đạo hàm bằng công thức (2.21) (xấp xỉ đạo hàm phải), sai số sẽ là $O(\varepsilon)$. Trong khi đó, nếu xấp xỉ đạo hàm bằng công thức (2.22) (xấp xỉ đạo hàm hai phía), sai số sẽ là $O(\varepsilon^2)$. Khi ε rất nhỏ, $O(\varepsilon^2) \ll O(\varepsilon)$, tức cách đánh giá sử dụng công thức 2.22 có sai số nhỏ hơn, và vì vậy nó được sử dụng nhiều hơn.

Chúng ta cũng có thể giải thích điều này bằng hình học.

Bằng hình học

Quan sát Hình 2.1, vector màu đỏ là đạo hàm *chính xác* của hàm số tại điểm có hoành độ bằng x_0 . Vector màu xanh lam và xanh lục lần lượt thể hiện cách xấp xỉ đạo hàm phía phải và phía trái. Vector màu nâu thể hiện cách xấp xỉ đạo hàm hai phía. Trong ba vector xấp xỉ đó, vector xấp xỉ hai phía màu nâu là gần với vector đỏ nhất nếu xét theo hướng.

Sự khác biệt giữa các cách xấp xỉ còn lớn hơn nữa nếu tại điểm x , hàm số bị *bẻ cong* mạnh hơn. Khi đó, xấp xỉ trái và phải sẽ khác nhau rất nhiều. Xấp xỉ hai bên sẽ *ổn định* hơn.

Từ đó ta thấy rằng xấp xỉ đạo hàm hai phía là xấp xỉ tốt hơn.

2.6.2 Xấp xỉ đạo hàm của hàm nhiều biến

Với hàm nhiều biến, công thức (2.22) được áp dụng cho từng biến khi các biến khác cố định. Cụ thể, ta sử dụng định nghĩa của hàm số nhận đầu vào là một ma trận như công thức (2.3). Mỗi thành phần của ma trận kết quả là đạo hàm của hàm số tại thành phần đó khi ta coi các thành phần còn lại cố định. Chúng ta sẽ thấy rõ điều này hơn ở cách lập trình so sánh hai cách tính đạo hàm ngay phía dưới.

Cách tính xấp xỉ đạo hàm theo phương pháp *numerical* thường cho giá trị khá chính xác. Tuy nhiên, cách này không được sử dụng để tính đạo hàm vì độ phức tạp quá cao so với cách tính trực tiếp. Tại mỗi thành phần, ta cần tính giá trị của hàm số tại phía trái và phía phải, như vậy sẽ không khả thi với các ma trận lớn. Khi so sánh đạo hàm *numerical* này với đạo hàm tính theo công thức, người ta thường giảm số chiều dữ liệu và giảm số điểm dữ liệu để thuận tiện cho tính toán. Nếu công thức đạo hàm ta tính được là chính xác, nó sẽ rất gần với đạo hàm *numerical*.

Đoạn Code 2.1 giúp kiểm tra đạo hàm của một hàm số khả vi $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$, có kèm theo hai ví dụ. Để sử dụng hàm kiểm tra `check_grad` này, ta cần viết hai hàm. Hàm thứ nhất là hàm `fn(X)` tính giá trị của hàm số tại X . Hàm thứ hai là hàm `gr(X)` tính giá trị của đạo hàm mà ta cần kiểm tra.

```

from __future__ import print_function
import numpy as np

def check_grad(fn, gr, X):
    X_flat = X.reshape(-1) # convert X to an 1d array -> 1 for loop needed
    shape_X = X.shape # original shape of X
    num_grad = np.zeros_like(X) # numerical grad, shape = shape of X
    grad_flat = np.zeros_like(X_flat) # 1d version of grad
    eps = 1e-6 # a small number, 1e-10 -> 1e-6 is usually good
    numElems = X_flat.shape[0] # number of elements in X
    # calculate numerical gradient
    for i in range(numElems): # iterate over all elements of X
        Xp_flat = X_flat.copy()
        Xn_flat = X_flat.copy()
        Xp_flat[i] += eps
        Xn_flat[i] -= eps
        Xp = Xp_flat.reshape(shape_X)
        Xn = Xn_flat.reshape(shape_X)
        grad_flat[i] = (fn(Xp) - fn(Xn)) / (2*eps)

    num_grad = grad_flat.reshape(shape_X)

    diff = np.linalg.norm(num_grad - gr(X))
    print('Difference between two methods should be small:', diff)

# ==== check if grad(trace(A*X)) == A^T ====
m, n = 10, 20
A = np.random.rand(m, n)
X = np.random.rand(n, m)

def fn1(X):
    return np.trace(A.dot(X))

def gr1(X):
    return A.T

check_grad(fn1, gr1, X)
# ==== check if grad(x^T*A*x) == (A + A^T)*x ====
A = np.random.rand(m, m)
x = np.random.rand(m, 1)

def fn2(x):
    return x.T.dot(A).dot(x)

def gr2(x):
    return (A + A.T).dot(x)

check_grad(fn2, gr2, x)

```

Code 2.1: Kiểm tra đạo hàm bằng phương pháp numerical.

Kết quả:

```
Difference between two methods should be small: 2.02303323394e-08  
Difference between two methods should be small: 2.10853872281e-09
```

Kết quả cho thấy sự khác nhau giữa Frobenious norm (mặc định của `np.linalg.norm`) của kết quả của hai cách tính là rất nhỏ. Sau khi chạy lại đoạn code với các giá trị `m`, `n` khác nhau và biến `x` khác nhau, nếu sự khác nhau vẫn là nhỏ, ta có thể tự tin rằng đạo hàm mà ta tính được là chính xác.

Bạn đọc có thể tự kiểm tra lại các công thức trong Bảng 2.1 theo phương pháp này.

Ôn tập Xác Suất

Chương này được viết dựa trên Chương 2 và 3 của cuốn *Computer Vision: Models, Learning, and Inference*—Simon J.D. Prince (<http://www.computervisionmodels.com>).

3.1 Xác Suất

3.1.1 Random variables

Một *biến ngẫu nhiên* (*random variable*) x là một đại lượng dùng để đo những đại lượng không xác định. Biến này có thể được dùng để ký hiệu kết quả/đầu ra (*outcome*) của một thí nghiệm, ví dụ như tung đồng xu, hoặc một đại lượng biến đổi trong tự nhiên, ví dụ như nhiệt độ trong ngày. Nếu chúng ta quan sát rất nhiều đầu ra $\{x_i\}_{i=1}^I$ của các thí nghiệm này, ta có thể nhận được những giá trị khác nhau ở mỗi thí nghiệm. Tuy nhiên, sẽ có những giá trị xảy ra nhiều lần hơn những giá trị khác, hoặc xảy ra gần một giá trị này hơn những giá trị khác. Thông tin về đầu ra này được đo bởi một *phân phối xác suất* (*probability distribution*) được biểu diễn bằng một hàm $p(x)$. Một biến ngẫu nhiên có thể là *rời rạc* (*discrete*) hoặc *liên tục* (*continuous*).

Một biến ngẫu nhiên rời rạc sẽ lấy giá trị trong một tập hợp các điểm rời rạc cho trước. Ví dụ tung đồng xu thì có hai khả năng là *head* và *tail*¹. Tập các giá trị này có thể là *có thứ tự* như khi tung xúc xắc hoặc *không có thứ tự*, ví dụ khi đầu ra là các giá trị *năng, mưa, bão*. Mỗi đầu ra có một giá trị xác suất tương ứng với nó. Các giá trị xác suất này không âm và có tổng bằng một.

$$\text{Nếu } x \text{ là biến ngẫu nhiên rời rạc thì } \sum_x p(x) = 1 \quad (3.1)$$

Biến ngẫu nhiên liên tục lấy các giá trị là các số thực. Những giá trị này có thể là hữu hạn, ví dụ thời gian làm bài của mỗi thí sinh trong một bài thi 180 phút, hoặc vô hạn, ví dụ thời

¹ đồng xu thường có một mặt có hình đầu người, được gọi là *head*, trái ngược với mặt này được gọi là mặt *tail*

gian phải chờ tới khách hàng tiếp theo. Không như biến ngẫu nhiên rời rạc, xác suất để đầu ra bằng *chính xác* một giá trị nào đó, theo lý thuyết, là bằng không. Thay vào đó, xác suất để đầu ra rơi vào một khoảng giá trị nào đó là khác không. Việc này được mô tả bởi *hàm mật độ xác suất* (*probability density function - pdf*). Hàm mật độ xác suất luôn cho giá trị dương, và tích phân của nó trên toàn miền giá trị đầu ra *possible outcome* phải bằng một.

$$\text{Nếu } x \text{ là biến ngẫu nhiên liên tục thì } \int p(x)dx = 1 \quad (3.2)$$

Nếu x là biến ngẫu nhiên rời rạc, thì $p(x) \leq 1, \forall x$. Trong khi đó, nếu x là biến ngẫu nhiên liên tục, $p(x)$ có thể nhận giá trị không âm bất kỳ, điều này vẫn đảm bảo là tích phân của hàm mật độ xác suất theo toàn bộ giá trị có thể có của x bằng một.

3.1.2 Xác suất đồng thời

Xét hai biến ngẫu nhiên x và y . Nếu ta quan sát rất nhiều cặp đầu ra của x và y , thì có những tổ hợp hai đầu ra xảy ra thường xuyên hơn những tổ hợp khác. Thông tin này được biểu diễn bằng một phân phối được gọi là *xác suất đồng thời* (*joint probability*) của x và y , được ký hiệu là $p(x, y)$, đọc là xác suất của x và y . Hai biến ngẫu nhiên x và y có thể đồng thời là biến ngẫu nhiên rời rạc, liên tục, hoặc một rời rạc, một liên tục. Luôn nhớ rằng tổng các xác suất trên mọi cặp giá trị có thể xảy ra (x, y) bằng một.

$$\text{Cả } x \text{ và } y \text{ là rời rạc: } \sum_{x,y} p(x, y) = 1 \quad (3.3)$$

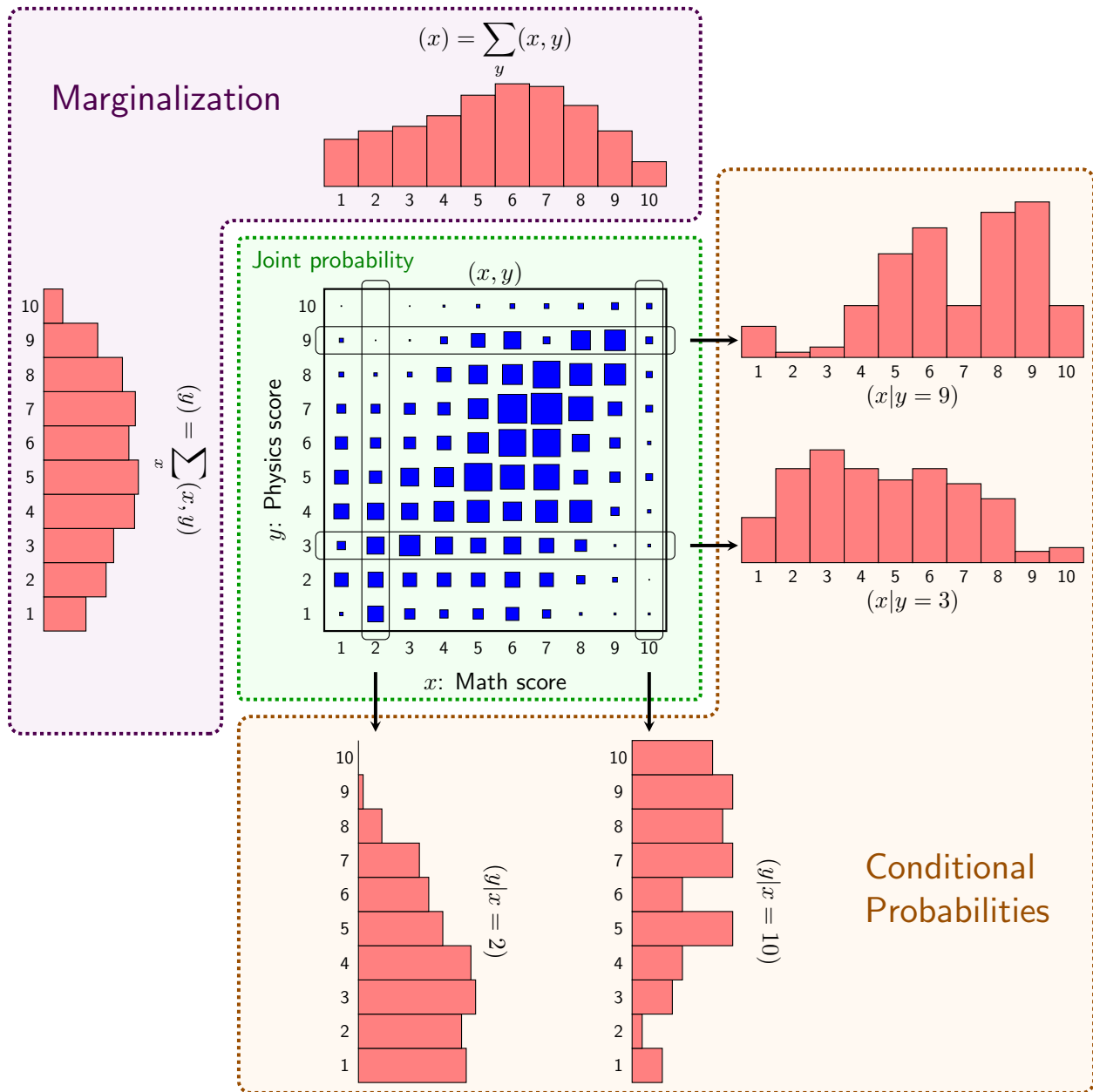
$$\text{Cả } x \text{ và } y \text{ là liên tục: } \int p(x, y)dxdy = 1 \quad (3.4)$$

$$x \text{ rời rạc, } y \text{ liên tục: } \sum_x \int p(x, y)dy = \int \left(\sum_x p(x, y) \right) dy = 1 \quad (3.5)$$

Xét ví dụ trong Hình 3.1, phần có nền màu lục nhạt. Biến ngẫu nhiên x thể hiện điểm thi môn Toán của học sinh ở một trường THPT trong một kỳ thi Quốc gia, biến ngẫu nhiên y thể hiện điểm thi môn Vật Lý cũng trong kỳ thi đó. Đại lượng $p(x = x^*, y = y^*)$ là tỉ lệ giữa tần suất số học sinh được *đồng thời* x^* điểm trong môn Toán và y^* điểm trong môn Vật Lý và toàn bộ số học sinh của trường đó. Tỉ lệ này có thể coi là xác suất khi số học sinh trong trường là lớn. Ở đây x^* và y^* là các số xác định. Thông thường, xác suất này được viết gọn lại thành $p(x^*, y^*)$, và $p(x, y)$ được dùng như một hàm tổng quát để mô tả các xác suất. Giả sử thêm rằng điểm các môn là các số tự nhiên từ 1 đến 10.

Các ô vuông màu lam thể hiện xác suất $p(x, y)$, với diện tích ô vuông càng to thể hiện xác suất đó càng lớn. Chú ý rằng tổng các xác suất này bằng một.

Các bạn có thể thấy rằng xác suất để một học sinh được 10 điểm môn Toán và 1 điểm môn Lý rất thấp, điều tương tự xảy ra với 10 điểm môn Lý và 1 điểm môn Toán. Ngược lại, xác suất để một học sinh được khoảng 7 điểm cả hai môn là cao nhất.



Hình 3.1: Xác suất đồng thời (phần trung tâm có nền màu lục nhạt), Xác suất biên (phía trên và bên trái) và Xác suất có điều kiện (phía dưới và bên phải).

Thông thường, chúng ta sẽ làm việc với các bài toán ở đó xác suất có điều kiện được xác định trên nhiều hơn hai biến ngẫu nhiên. Chẳng hạn, $p(x, y, z)$ thể hiện joint probability của ba biến ngẫu nhiên x, y và z . Khi có nhiều biến ngẫu nhiên, ta có thể viết chúng dưới dạng vector. Cụ thể, ta có thể viết $p(\mathbf{x})$ để thể hiện xác suất có điều kiện của biến ngẫu nhiên nhiều chiều $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$. Khi có nhiều tập các biến ngẫu nhiên, ví dụ \mathbf{x} và \mathbf{y} , ta có thể biết $p(\mathbf{x}, \mathbf{y})$ để thể hiện xác suất có điều kiện của tất cả các thành phần trong hai biến ngẫu nhiên nhiều chiều này.

3.1.3 Xác suất biên

Nếu biết xác suất đồng thời của nhiều biến ngẫu nhiên, ta cũng có thể xác định được phân phối xác suất của từng biến bằng cách lấy tổng với biến ngẫu nhiên rời rạc hoặc tích phân với biến ngẫu nhiên liên tục theo tất cả các biến còn lại:

$$\text{Nếu } x, y \text{ rời rạc : } p(x) = \sum_y p(x, y) \quad (3.6)$$

$$p(y) = \sum_x p(x, y) \quad (3.7)$$

$$\text{Nếu } x, y \text{ liên tục : } p(x) = \int p(x, y) dy \quad (3.8)$$

$$p(y) = \int p(x, y) dx \quad (3.9)$$

Với nhiều biến hơn, chẳng hạn bốn biến rời rạc x, y, z, w , cách tính được thực hiện tương tự:

$$p(x) = \sum_{y,z,w} p(x, y, z, w) \quad (3.10)$$

$$p(x, y) = \sum_{z,w} p(x, y, z, w) \quad (3.11)$$

Cách xác định xác suất của một biến dựa trên xác suất đồng thời của nó với các biến khác được gọi là *marginalization*. Phân phối đó được gọi là *xác suất biên* (*marginal probability*).

Từ đây trở đi, nếu không đề cập gì thêm, chúng ta sẽ dùng ký hiệu \sum để chỉ chung cho cả hai loại biến. Nếu biến ngẫu nhiên là liên tục, bạn đọc ngầm hiểu rằng dấu \sum cần được thay bằng dấu tích phân \int , biến lấy vi phân chính là biến được viết dưới dấu \sum . Chẳng hạn, trong (3.11), nếu z là liên tục, w là rời rạc, công thức đúng sẽ là

$$p(x, y) = \sum_w \left(\int p(x, y, z, w) dz \right) = \int \left(\sum_w p(x, y, z, w) \right) dz \quad (3.12)$$

Quay lại ví dụ trong Hình 3.1 với hai biến ngẫu nhiên rời rạc x, y . Lúc này, $p(x)$ được hiểu là xác suất để một học sinh đạt được x điểm môn Toán. Xác suất này được thể hiện ở khu vực có nền màu tím nhạt, phía trên. Nhắc lại rằng xác suất ở đây thực ra là tỉ lệ giữa số học sinh đạt x điểm môn Toán và toàn bộ số học sinh. Có hai cách tính xác suất này. Cách thứ nhất, dựa trên cách vừa định nghĩa, là đếm số học sinh được x điểm môn toán rồi chia cho tổng số học sinh. Cách tính thứ hai dựa trên xác suất đồng thời đã biết về xác suất để một học sinh được x điểm môn Toán và y điểm môn Lý. Số lượng học sinh đạt $x = x^*$ điểm môn Toán sẽ bằng tổng số lượng học sinh đạt $x = x^*$ điểm môn Toán và y điểm môn Lý, với y là một giá trị bất kỳ từ 1 đến 10. vì vậy, để tính xác suất $p(x)$, ta chỉ cần tính tổng của toàn bộ $p(x, y)$ với y chạy từ 1 đến 10. Tương tự nếu ta muốn tính $p(y)$ (xem phần bên trái của khu vực nền tím nhạt).

Dựa trên nhận xét này, mỗi giá trị của $p(x)$ chính bằng tổng các giá trị trong cột thứ x của hình vuông trung tâm nền xanh lục. Mỗi giá trị của $p(y)$ sẽ bằng tổng các giá trị trong hàng thứ y tính từ dưới lên. Chú ý rằng tổng các xác suất luôn bằng một.

3.1.4 Xác suất có điều kiện.

Dựa vào phân phối điểm của các học sinh, liệu ta có thể tính được xác suất để một học sinh được điểm 10 môn Lý, biết rằng học sinh đó được điểm 1 môn Toán?

Xác suất để một biến ngẫu nhiên x nhận một giá trị nào đó biết rằng biến ngẫu nhiên y có giá trị y^* được gọi là *xác suất có điều kiện* (*conditional probability*), được ký hiệu là $p(x|y = y^*)$.

Xác suất có điều kiện $p(x|y = y^*)$ có thể được tính dựa trên xác suất đồng thời $p(x, y)$. Quay lại Hình 3.1 với vùng có nền màu nâu nhạt. Nếu biết rằng $y = 9$, xác suất $p(x|y = 9)$ có thể tính được dựa trên hàng thứ chín của hình vuông trung tâm, tức hàng $p(x, y = 9)$. Trong hàng này, những ô vuông lớn hơn thể hiện xác suất lớn hơn. Tương ứng như thế, $p(x|y = 9)$ cũng lớn nếu $p(x, y = 9)$ lớn. Chú ý rằng tổng các xác suất $\sum_x p(x, y = 9)$ nhỏ hơn một, và bằng tổng các xác suất trên hàng thứ chín này. Để thoả mãn điều kiện tổng các xác suất bằng một, ta cần chia mỗi đại lượng $p(x, y = 9)$ cho tổng của toàn hàng này. Tức là

$$p(x|y = 9) = \frac{p(x, y = 9)}{\sum_x p(x, y = 9)} = \frac{p(x, y = 9)}{p(y = 9)} \quad (3.13)$$

Tổng quát,

$$p(x|y = y^*) = \frac{p(x, y = y^*)}{\sum_x p(x, y = y^*)} = \frac{p(x, y = y^*)}{p(y = y^*)} \quad (3.14)$$

ở đây ta đã sử dụng công thức tính xác suất biên trong (3.7) cho mẫu số. Thông thường, ta có thể viết xác suất có điều kiện mà không cần chỉ rõ giá trị $y = y^*$ và có công thức gọn hơn:

$$p(x|y) = \frac{p(x, y)}{p(y)}, \text{ và tương tự, } p(y|x) = \frac{p(y, x)}{p(x)} \quad (3.15)$$

Từ đó ta có quan hệ

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x) \quad (3.16)$$

Khi có nhiều hơn hai biến ngẫu nhiên, ta có các công thức

$$p(x, y, z, w) = p(x, y, z|w)p(w) \quad (3.17)$$

$$= p(x, y|z, w)p(z, w) = p(x, y|z, w)p(z|w)p(w) \quad (3.18)$$

$$= p(x|y, z, w)p(y|z, w)p(z|w)p(w) \quad (3.19)$$

Công thức (3.19) có dạng *chuỗi* (*chain*) và được sử dụng nhiều sau này.

3.1.5 Quy tắc Bayes

Công thức (3.16) biểu diễn xác suất đồng thời theo hai cách. Từ đó ta có thể suy ra:

$$p(y|x)p(x) = p(x|y)p(y) \quad (3.20)$$

Biến đổi một chút:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \quad (3.21)$$

$$= \frac{p(x|y)p(y)}{\sum_y p(x,y)} \quad (3.22)$$

$$= \frac{p(x|y)p(y)}{\sum_y p(x|y)p(y)} \quad (3.23)$$

ở đó dòng thứ hai và thứ ba các công thức về xác suất biên và xác suất đồng thời ở mẫu số đã được sử dụng. Từ (3.23) ta có thể thấy rằng $p(y|x)$ hoàn toàn có thể tính được nếu ta biết mọi $p(x|y)$ và $p(y)$. Tuy nhiên, việc tính trực tiếp xác suất này thường là phức tạp.

Ba công thức (3.21)-(3.23) thường được gọi là Quy tắc Bayes (Bayes' rule). Chúng được sử dụng rộng rãi trong Machine Learning

3.1.6 Biến ngẫu nhiên độc lập

Nếu biết giá trị của một biến ngẫu nhiên x không mang lại thông tin về việc suy ra giá trị của biến ngẫu nhiên y (và ngược lại), thì ta nói rằng hai biến ngẫu nhiên là *độc lập* (*independent*). Chẳng hạn, chiều cao của một học sinh và điểm thi môn Toán của học sinh đó có thể coi là hai biến ngẫu nhiên *độc lập*.

Khi hai biến ngẫu nhiên x và y là *độc lập*, ta sẽ có:

$$p(x|y) = p(x) \quad (3.24)$$

$$p(y|x) = p(y) \quad (3.25)$$

Thay vào biểu thức xác suất đồng thời trong (3.16), ta có:

$$p(x,y) = p(x|y)p(y) = p(x)p(y) \quad (3.26)$$

3.1.7 Kỳ vọng và ma trận hiệp phương sai

Kỳ vọng (*expectation*) của một biến ngẫu nhiên được định nghĩa là

$$E[x] = \sum_x xp(x) \quad \text{nếu } x \text{ là rời rạc} \quad (3.27)$$

$$E[x] = \int xp(x)dx \quad \text{nếu } x \text{ là liên tục} \quad (3.28)$$

Giả sử $f(\cdot)$ là một hàm số trả về một số với mỗi giá trị x^* của biến ngẫu nhiên x . Khi đó, nếu x là biến ngẫu nhiên rời rạc, ta sẽ có

$$E[f(x)] = \sum_x f(x)p(x) \quad (3.29)$$

Công thức cho biến ngẫu nhiên liên tục cũng được viết tương tự.

Với xác suất đồng thời

$$E[f(x, y)] = \sum_{x, y} f(x, y)p(x, y)dxdy \quad (3.30)$$

Có ba tính chất cần nhớ về kỳ vọng:

1. Kỳ vọng của một hằng số theo một biến ngẫu nhiên x bất kỳ bằng chính hằng số đó:

$$E[\alpha] = \alpha \quad (3.31)$$

2. Kỳ vọng có tính chất tuyến tính:

$$E[\alpha x] = \alpha E[x] \quad (3.32)$$

$$E[f(x) + g(x)] = E[f(x)] + E[g(x)] \quad (3.33)$$

3. Kỳ vọng của tích hai biến ngẫu nhiên bằng tích kỳ vọng của hai biến đó **nếu hai biến ngẫu nhiên đó là độc lập**.

$$E[f(x)g(y)] = E[f(x)]E[g(y)] \quad (3.34)$$

Khái niệm kỳ vọng thường đi kèm với khái niệm *phương sai* (*variance*) trong không gian một chiều, và *ma trận hiệp phương sai* (*covariance matrix*) trong không gian nhiều chiều.

Với dữ liệu một chiều

Cho N giá trị x_1, x_2, \dots, x_N . Kỳ vọng và phương sai của bộ dữ liệu này được tính theo công thức:

$$\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n = \frac{1}{N} \mathbf{x} \mathbf{1} \quad (3.35)$$

$$\sigma^2 = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})^2 \quad (3.36)$$

với $\mathbf{x} = [x_1, x_2, \dots, x_N]$, và $\mathbf{1} \in \mathbb{R}^N$ là vector cột chứa toàn phần tử 1. Kỳ vọng đơn giản là trung bình cộng của toàn bộ các giá trị. Phương sai là trung bình cộng của bình phương khoảng cách từ mỗi điểm tới kỳ vọng. Phương sai càng nhỏ thì các điểm dữ liệu càng gần với kỳ vọng, tức các điểm dữ liệu càng giống nhau. Phương sai càng lớn thì ta nói dữ liệu càng có tính phân tán. Ví dụ về kỳ vọng và phương sai của dữ liệu một chiều có thể được thấy trong Hình 3.2a. Căn bậc hai của phương sai, σ còn được gọi là *độ lệch chuẩn* (*standard deviation*) của dữ liệu.

Với dữ liệu nhiều chiều

Cho N điểm dữ liệu được biểu diễn bởi các vector cột $\mathbf{x}_1, \dots, \mathbf{x}_N$, khi đó, *vector kỳ vọng* và *ma trận hiệp phương sai* của toàn bộ dữ liệu được định nghĩa là:

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \quad (3.37)$$

$$\mathbf{S} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \bar{\mathbf{x}})(\mathbf{x}_n - \bar{\mathbf{x}})^T = \frac{1}{N} \hat{\mathbf{X}} \hat{\mathbf{X}}^T \quad (3.38)$$

Trong đó $\hat{\mathbf{X}}$ được tạo bằng cách trừ mỗi cột của \mathbf{X} đi $\bar{\mathbf{x}}$:

$$\hat{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}} \quad (3.39)$$

Một vài tính chất của ma trận hiệp phương sai:

- Ma trận hiệp phương sai là một ma trận đối xứng, hơn nữa, nó là một ma trận **nửa xác định dương**.
- Mọi phần tử trên đường chéo của ma trận hiệp phương sai là các số không âm. Chúng cũng chính là phương sai của từng chiều của dữ liệu.
- Các phần tử ngoài đường chéo $s_{ij}, i \neq j$ thể hiện sự tương quan giữa thành phần thứ i và thứ j của dữ liệu, còn được gọi là hiệp phương sai. Giá trị này có thể dương, âm hoặc bằng không. Khi nó bằng không, ta nói rằng hai thành phần i, j trong dữ liệu là *không tương quan* (*uncorrelated*).
- Nếu ma trận hiệp phương sai là ma trận đường chéo, ta có dữ liệu hoàn toàn không tương quan giữa các chiều.

Ví dụ về dữ liệu không tương quan và tương quan được cho trong Hình 3.2b và 3.2c.

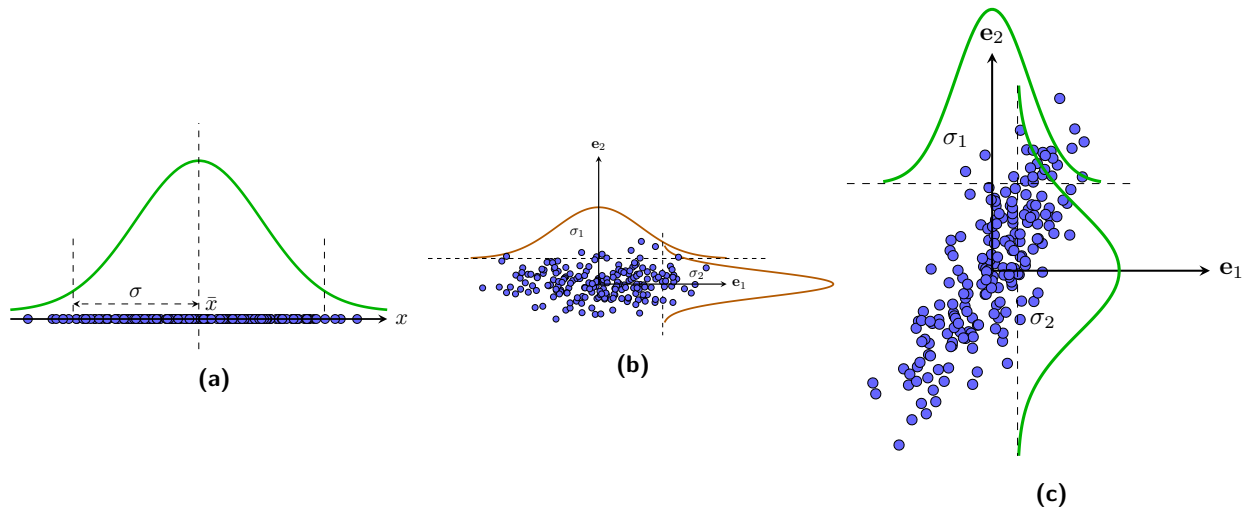
3.2 Một vài phân phối thường gặp

3.2.1 Phân phối Bernoulli

Phân phối Bernoulli là một phân phối rời rạc mô tả các biến ngẫu nhiên nhị phân: trường hợp đầu ra chỉ nhận một trong hai giá trị $x \in \{0, 1\}$. Hai giá trị này có thể là *head* và *tail* khi tung đồng xu; có thể là *giao dịch lừa đảo* và *giao dịch thông thường* trong bài toán xác định giao dịch lừa đảo trong tín dụng; có thể là *người* và *không phải người* trong bài toán tìm xem trong một bức ảnh có người hay không.

Bernoulli distribution được mô tả bằng một tham số $\lambda \in [0, 1]$ và là xác suất để biến ngẫu nhiên $x = 1$. Xác suất của mỗi đầu ra sẽ là

$$p(x = 1) = \lambda, \quad p(x = 0) = 1 - p(x = 1) = 1 - \lambda \quad (3.40)$$



Hình 3.2: Ví dụ về kỳ vọng và phương sai. (a) Trong không gian một chiều. (b) Trong không gian hai chiều mà hai chiều không tương quan. Trong trường hợp này, ma trận hiệp phương sai là ma trận đường chéo với hai phần tử trên đường chéo là σ_1, σ_2 , đây cũng chính là hai trị riêng của ma trận hiệp phương sai và là phương sai của mỗi chiều dữ liệu. (c) Dữ liệu trong không gian hai chiều có tương quan. Theo mỗi chiều, ta có thể tính được kỳ vọng và phương sai. Phương sai càng lớn thì dữ liệu trong chiều đó càng phân tán. Trong ví dụ này, dữ liệu theo chiều thứ hai phân tán nhiều hơn so với chiều thứ nhất.

Hai đẳng thức này thường được viết gọn lại:

$$p(x) = \lambda^x(1 - \lambda)^{1-x} \tag{3.41}$$

với giả định rằng $0^0 = 1$. Thật vậy, $p(0) = \lambda^0(1 - \lambda)^1 = 1 - \lambda$, và $p(1) = \lambda^1(1 - \lambda)^0 = \lambda$.

Phân phối Bernoulli thường được ký hiệu ngắn gọn dưới dạng

$$p(x) = \text{Bern}_x[\lambda] \tag{3.42}$$

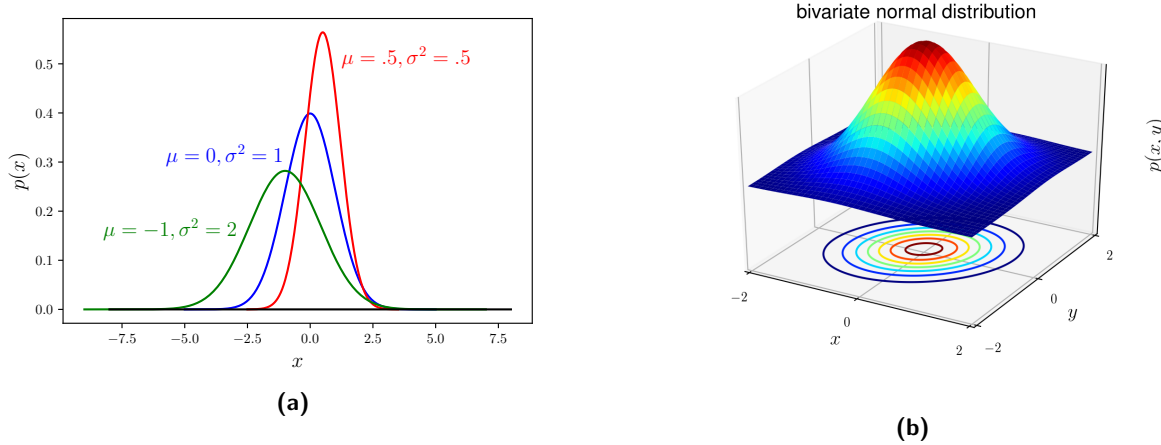
3.2.2 Phân phối Categorical

Trong nhiều trường hợp, đầu ra của biến ngẫu nhiên rời rạc có thể là một trong nhiều hơn hai giá trị khác nhau. Ví dụ, một bức ảnh có thể chứa một chiếc xe, một người, hoặc một con mèo. Khi đó, ta dùng một phân phối tổng quát của phân phối Bernoulli, được gọi là *phân phối Categorical*. Các đầu ra được mô tả bởi một phần tử trong tập hợp $\{1, 2, \dots, K\}$.

Nếu có K đầu ra, phân phối Categorical sẽ được mô tả bởi K tham số, viết dưới dạng vector: $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_K]$ với các λ_k không âm và có tổng bằng một. Mỗi giá trị λ_k thể hiện xác suất để đầu ra nhận giá trị k : $p(x = k) = \lambda_k$.

Phân phối Categorical thường được ký hiệu dưới dạng:

$$p(x) = \text{Cat}_x[\lambda] \tag{3.43}$$



Hình 3.3: Ví dụ về hàm mật độ xác suất của (a) phân phối chuẩn một chiều, và (b) phân phối chuẩn hai chiều.

Nếu thay vì biểu diễn đầu ra là một số k trong tập hợp $\{1, 2, \dots, K\}$, ta biểu diễn đầu ra là một vector ở dạng *one-hot*, tức một vector K phần tử với chỉ phần tử thứ k bằng một, các phần tử còn lại bằng không. Nói cách khác, tập hợp các đầu ra là tập hợp các vector đơn vị bậc K : $\mathbf{x} \in \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_K\}$ với \mathbf{e}_k là vector đơn vị thứ k . Khi đó, ta sẽ có

$$p(\mathbf{x} = \mathbf{e}_k) = \prod_{j=1}^K \lambda_j^{x_j} = \lambda_k \quad (3.44)$$

Khi $\mathbf{x} = \mathbf{e}_k$, $x_k = 1$, $x_j = 0$, $\forall j \neq k$. Thay vào (3.44) ta sẽ được $p(\mathbf{x} = \mathbf{e}_k) = \lambda_k = p(x = k)$.

3.2.3 Phân phối chuẩn một chiều

Phân phối chuẩn một chiều (*univariate normal* hoặc *Gaussian distribution*) được định nghĩa trên các biến liên tục nhận giá trị $x \in (-\infty, \infty)$. Đây là một phân phối được sử dụng nhiều nhất với các biến ngẫu nhiên liên tục. Phân phối này được mô tả bởi hai tham số: *kỳ vọng* μ và *phương sai* (*variance*) σ^2 . Giá trị μ có thể là bất kỳ số thực nào, thể hiện vị trí của giá trị mà tại đó mà hàm mật độ xác suất đạt giá trị cao nhất. Giá trị σ^2 là một giá trị dương, với σ thể hiện *độ rộng* của phân phối này. σ lớn chứng tỏ khoảng giá trị đầu ra có khoảng biến đổi mạnh, và ngược lại.

Hàm mật độ xác suất của phân phối này được định nghĩa là

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) \quad (3.45)$$

Hoặc được viết gọn hơn dưới dạng $p(x) = \text{Norm}_x[\mu, \sigma^2]$, hoặc $\mathcal{N}(\mu, \sigma^2)$.

Ví dụ về đồ thị hàm mật độ xác suất của phân phối chuẩn một chiều được cho trên Hình 3.3a.

3.2.4 Phân phối chuẩn nhiều chiều

Phân phối này là trường hợp tổng quát của phân phối chuẩn khi biến ngẫu nhiên là nhiều chiều, giả sử là D chiều. Có hai tham số mô tả phân phối này: *vector kỳ vọng* $\boldsymbol{\mu} \in \mathbb{R}^D$ và *ma trận hiệp phương sai* $\boldsymbol{\Sigma} \in \mathbb{S}^D$ là một ma trận *đối xứng xác định dương*.

Hàm mật độ xác suất có dạng

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{D/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right) \quad (3.46)$$

với $|\boldsymbol{\Sigma}|$ là định thức của ma trận hiệp phương sai $\boldsymbol{\Sigma}$.

Phân phối này thường được viết gọn lại dưới dạng $p(\mathbf{x}) = \text{Norm}_{\mathbf{x}}[\boldsymbol{\mu}, \boldsymbol{\Sigma}]$, hoặc $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$.

Ví dụ về hàm mật độ xác suất của một phân phối chuẩn hai chiều (*bivariate normal distribution*) được mô tả bởi một mặt cong cho trên Hình 3.3b. Nếu cắt mặt này theo các mặt phẳng song song với mặt đáy, ta sẽ thu được các hình ellipse đồng tâm.

3.2.5 Phân phối Beta

Phân phối Beta (*Beta distribution*) là một phân phối liên tục được định nghĩa trên một biến ngẫu nhiên $\lambda \in [0, 1]$ Phân phối Beta distribution được dùng để mô tả *tham số* cho một distribution khác. Cụ thể, phân phối này phù hợp với việc mô tả sự *biến động* của tham số λ trong phân phối Bernoulli.

Phân phối Beta được mô tả bởi hai tham số *dương* α, β . Hàm mật độ xác suất của nó là

$$p(\lambda) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \lambda^{\alpha-1} (1 - \lambda)^{\beta-1} \quad (3.47)$$

với $\Gamma(\cdot)$ là hàm số gamma, được định nghĩa là

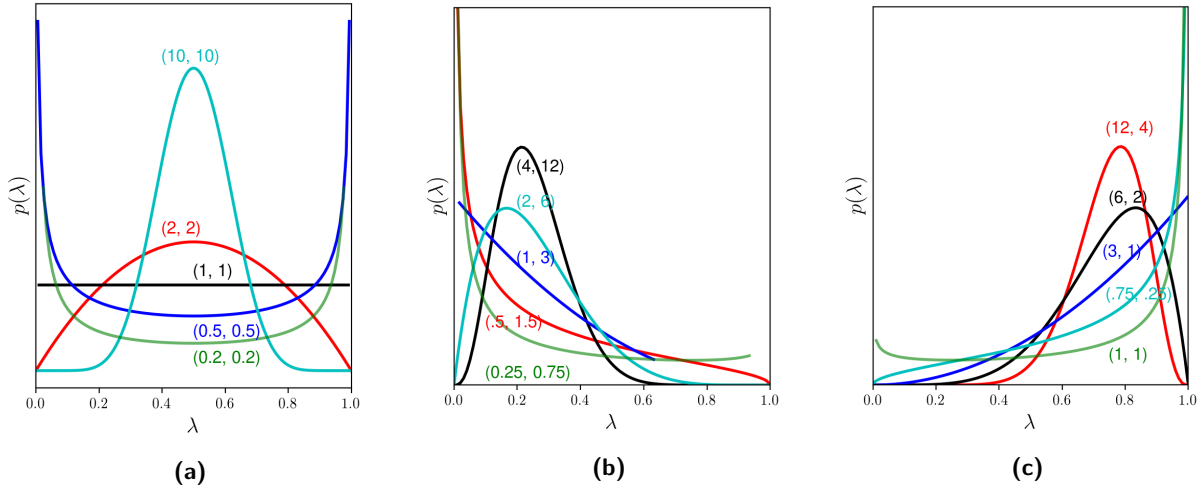
$$\Gamma(z) = \int_0^{\infty} t^{z-1} \exp(-t) dt \quad (3.48)$$

Trên thực tế, việc tính giá trị của hàm số gamma không thực sự quan trọng vì nó chỉ mang tính chuẩn hoá để tổng xác suất bằng một.

Dạng gọn của phân phối Beta: $p(\lambda) = \text{Beta}_{\lambda}[\alpha, \beta]$

Hình 3.4 minh hoạ các hàm mật độ xác suất của phân phối Beta với các cặp giá trị (α, β) khác nhau.

- Trong Hình 3.4a, khi $\alpha = \beta$. Đồ thị của các hàm mật độ xác suất đối xứng qua đường thẳng $\lambda = 0.5$. Khi $\alpha = \beta = 1$, thay vào (3.47) ta thấy $p(\lambda) = 1$ với mọi λ . Trong trường



Hình 3.4: Ví dụ về hàm mật độ xác suất của phân phối Beta. (a) $\alpha = \beta$, đồ thị hàm số là đối xứng. (b) $\alpha < \beta$, đồ thị hàm số lệch sang trái, chứng tỏ xác suất λ nhỏ là lớn. (c) $\alpha > \beta$, đồ thị hàm số lệch sang phải, chứng tỏ xác suất λ lớn là lớn.

hợp này, phân phối Beta trở thành *phân phối đều* (*uniform distribution*). Khi $\alpha = \beta > 1$, các hàm số đạt giá trị cao tại gần trung tâm, tức là khả năng cao là λ sẽ nhận giá trị xung quanh điểm 0.5. Khi $\alpha = \beta < 1$, hàm số đạt giá trị cao tại các điểm gần 0 và 1.

- Trong Hình 3.4b, khi $\alpha < \beta$, ta thấy rằng đồ thị có xu hướng lệch sang bên trái. Các giá trị (α, β) này nên được sử dụng nếu ta dự đoán rằng λ là một số nhỏ hơn 0.5.
- Trong Hình 3.4c, khi $\alpha > \beta$, điều ngược lại xảy ra với các hàm số đạt giá trị cao tại các điểm gần 1.

3.2.6 Phân phối Dirichlet

Phân phối Dirichlet chính là trường hợp tổng quát của phân phối Beta khi được dùng để mô tả tham số của phân phối Categorical. Nhắc lại rằng phân phối Categorical là trường hợp tổng quát của phân phối Bernoulli.

Phân phối Dirichlet được định nghĩa trên K biến liên tục $\lambda_1, \dots, \lambda_K$ trong đó các λ_k không âm và có tổng bằng một. Bởi vậy, nó phù hợp để mô tả tham số của phân phối Categorical. Có K tham số *dương* để mô tả một phân phối Dirichlet: $\alpha_1, \dots, \alpha_K$.

Hàm mật độ xác suất của phân phối Dirichlet là

$$p(\lambda_1, \dots, \lambda_K) = \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K \lambda_k^{\alpha_k - 1} \quad (3.49)$$

Cách biểu diễn ngắn gọn: $p(\lambda_1, \dots, \lambda_K) = \text{Dir}_{\lambda_1, \dots, \lambda_K}[\alpha_1, \dots, \alpha_K]$

Maximum Likelihood và Maximum A Posteriori

4.1 Giới thiệu

Có rất nhiều mô hình machine learning được xây dựng dựa trên các mô hình thống kê (*statistical models*). Các mô hình thống kê thường dựa trên các phân phối xác suất đã được đề cập trong Chương 3. Với phân phối Bernoulli, tham số là biến λ . Với phân phối chuẩn nhiều chiều, các tham số là mean vector μ và ma trận hiệp phương sai Σ . Với một mô hình thống kê bất kỳ, ký hiệu θ là tập hợp tất cả các tham số của mô hình đó. Learning chính là quá trình *ước lượng* (*estimate*) bộ tham số θ sao cho mô hình tìm được khớp với phân phối của dữ liệu nhất. Quá trình này còn được gọi là *ước lượng tham số* (*parameter estimation*).

Có hai cách ước lượng tham số thường được dùng trong các mô hình machine learning thống kê. Cách thứ nhất chỉ dựa trên dữ liệu đã biết trong tập huấn luyện, được gọi là *maximum likelihood estimation* hay *ML estimation* hoặc *MLE*. Cách thứ hai không những dựa trên tập huấn luyện mà còn dựa trên những thông tin biết trước của các tham số. Những thông tin này có thể có được bằng *cảm quan* của người xây dựng mô hình. *Cảm quan* càng rõ ràng, càng hợp lý thì khả năng thu được bộ tham số tốt là càng cao. Chẳng hạn, thông tin biết trước của λ trong Bernoulli distribution là việc nó là một số trong đoạn $[0, 1]$. Với bài toán tung đồng xu, với λ là xác suất có được mặt *head*, ta dự đoán được rằng giá trị này nên là một số gần với 0.5. Cách ước lượng tham số thứ hai này được gọi là *maximum a posteriori estimation* hay *MAP estimation*. Trong chương này, chúng ta cùng tìm hiểu ý tưởng và cách giải quyết bài toán ước lượng tham số mô hình theo *MLE* hoặc *MAP Estimation*.

4.2 Maximum likelihood estimation

4.2.1 Ý tưởng

Giả sử có các điểm dữ liệu $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$. Giả sử thêm rằng ta đã biết các điểm dữ liệu này tuân theo một phân phối nào đó được mô tả bởi bộ tham số θ .

Maximum likelihood estimation là việc đi tìm bộ tham số θ sao cho xác suất sau đây đạt giá trị lớn nhất:

$$\theta = \max_{\theta} p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta) \quad (4.1)$$

Biểu thức (4.1) có ý nghĩa như thế nào và vì sao việc này có lý?

Giả sử rằng ta đã biết dạng của mô hình, và mô hình này được mô tả bởi bộ tham số θ . Như vậy, $p(\mathbf{x}_1 | \theta)$ chính là xác suất xảy ra sự kiện \mathbf{x}_1 biết rằng mô hình được mô tả bởi bộ tham số θ (đây là một xác suất có điều kiện). Và $p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta)$ chính là xác suất để toàn bộ các sự kiện $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$ đồng thời xảy ra, xác suất đồng thời này còn được gọi là *likelihood*. Ở đây, *likelihood* chính là hàm mục tiêu.

Bởi vì sự việc đã xảy ra, tức dữ liệu huấn luyện bản thân chúng đã như thế, xác suất đồng thời này cần phải càng cao càng tốt. Việc này cũng giống như việc đã biết *kết quả*, và ta cần đi tìm *nguyên nhân* sao cho xác suất xảy ra kết quả càng cao càng tốt. MLE chính là việc đi tìm bộ tham số θ sao cho Likelihood là lớn nhất. Trong mô hình này ta cũng có một bài toán tối ưu với hàm mục tiêu là $p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta)$. Lúc này ta không tối thiểu hàm mục tiêu mà cần tối đa nó, vì ta muốn rằng xác suất xảy ra việc này là lớn nhất.

4.2.2 Giả sử về sự độc lập và log-likelihood

Việc giải trực tiếp bài toán (4.1) thường là phức tạp vì việc đi tìm mô hình xác suất đồng thời cho toàn bộ dữ liệu là ít khi khả thi. Một cách tiếp cận phổ biến là giả sử đơn giản rằng các điểm dữ liệu \mathbf{x}_n là độc lập với nhau. Nói cách khác, ta xấp xỉ likelihood trong (4.1) bởi

$$p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta) \approx \prod_{n=1}^N p(\mathbf{x}_n | \theta) \quad (4.2)$$

(Nhắc lại rằng hai sự kiện x, y là độc lập nếu xác suất đồng thời của chúng bằng tích xác suất của từng sự kiện: $p(x, y) = p(x)p(y)$. Và khi là xác suất có điều kiện: $p(x, y | z) = p(x | z)p(y | z)$.) Lúc đó, bài toán (4.1) có thể được giải quyết bằng cách giải bài toán tối ưu sau:

$$\theta = \max_{\theta} \prod_{n=1}^N p(\mathbf{x}_n | \theta) \quad (3) \quad (4.3)$$

Việc tối ưu một tích thường phức tạp hơn việc tối ưu một tổng, vì vậy việc tối đa hàm mục tiêu thường được chuyển về việc tối đa log của hàm mục tiêu:

$$\theta = \max_{\theta} \sum_{n=1}^N \log(p(\mathbf{x}_n | \theta)) \quad (4.4)$$

Ôn lại một chút về hai tính chất của hàm logarit: (i) log của một tích bằng tổng của các log, và (ii) vì log là một hàm đồng biến, một biểu thức dương sẽ là lớn nhất nếu log của nó là lớn nhất, và ngược lại.

4.2.3 Ví dụ

Ví dụ 1: phân phối Bernoulli

Bài toán: giả sử tung một đồng xu N lần và nhận được n mặt *head*. Ước lượng xác suất khi tung đồng xu nhận được mặt *head*.

Lời giải:

Một cách trực quan, ta có thể ước lượng được rằng xác suất đó chính là $\lambda = \frac{n}{N}$. Chúng ta cũng ước lượng giá trị này sử dụng MLE.

Giả sử λ là xác suất để nhận được một mặt *head*. Đặt x_1, x_2, \dots, x_N là các đầu ra nhận được, trong đó có n giá trị bằng 1 tương ứng với mặt *head* và $m = N - n$ giá trị bằng 0 tương ứng với mặt *tail*. Ta có thể suy ra ngay rằng

$$\sum_{i=1}^N x_i = n, \quad N - \sum_{i=1}^N x_i = N - n = m \quad (4.5)$$

Vì đây là một xác suất của biến ngẫu nhiên nhị phân rời rạc, ta có thể nhận thấy việc nhận được mặt *head* hay *tail* khi tung đồng xu tuân theo phân phối Bernoulli:

$$p(x_i|\lambda) = \lambda^{x_i}(1 - \lambda)^{1-x_i} \quad (4.6)$$

Khi đó tham số mô hình λ có thể được ước lượng bằng việc giải bài toán tối ưu sau đây, với giả sử rằng kết quả của các lần tung đồng xu là độc lập với nhau:

$$\lambda = \operatorname{argmax}_{\lambda} [p(x_1, x_2, \dots, x_N|\lambda)] = \operatorname{argmax}_{\lambda} \left[\prod_{i=1}^N p(x_i|\lambda) \right] \quad (4.7)$$

$$= \operatorname{argmax}_{\lambda} \left[\prod_{i=1}^N \lambda^{x_i}(1 - \lambda)^{1-x_i} \right] = \operatorname{argmax}_{\lambda} \left[\lambda^{\sum_{i=1}^N x_i} (1 - \lambda)^{N - \sum_{i=1}^N x_i} \right] \quad (4.8)$$

$$= \operatorname{argmax}_{\lambda} [\lambda^n (1 - \lambda)^m] = \operatorname{argmax}_{\lambda} [n \log(\lambda) + m \log(1 - \lambda)] \quad (4.9)$$

trong (4.9), ta đã lấy log của hàm mục tiêu. Tới đây, bài toán tối ưu (4.9) có thể được giải bằng cách lấy đạo hàm của hàm mục tiêu bằng 0. Tức λ là nghiệm của phương trình

$$\frac{n}{\lambda} - \frac{m}{1 - \lambda} = 0 \Leftrightarrow \frac{n}{\lambda} = \frac{m}{1 - \lambda} \Leftrightarrow \lambda = \frac{n}{n + m} = \frac{n}{N} \quad (4.10)$$

Vậy kết quả ta ước lượng ban đầu là có cơ sở.

Ví dụ 2: Categorical distribution

Một ví dụ khác phức tạp hơn một chút.

Bài toán: giả sử tung một viên xúc sắc sáu mặt có xác suất rơi vào các mặt có thể không đều nhau. Giả sử trong N lần tung, số lượng xuất hiện các mặt thứ nhất, thứ hai, ..., thứ sáu lần lượt là n_1, n_2, \dots, n_6 lần với $\sum_{i=1}^6 n_i = N$. Tính xác suất rơi vào mỗi mặt ở lần tung tiếp theo. Giả sử thêm rằng $n_i > 0, \forall i = 1, \dots, 6$.

Lời giải:

Bài toán này có vẻ phức tạp hơn bài toán trên một chút, nhưng ta cũng có thể dự đoán được ước lượng tốt nhất của xác suất rơi vào mặt thứ i là $\lambda_i = \frac{n_i}{N}$.

Mã hoá mỗi quan sát đầu ra thứ i bởi một vector 6 chiều $\mathbf{x}_i \in \{0, 1\}^6$ trong đó các phần tử của nó bằng 0 trừ phần tử tương ứng với mặt quan sát được là bằng 1. Nhận thấy rằng $\sum_{i=1}^N x_i^j = n_j, \forall j = 1, 2, \dots, 6$, trong đó x_i^j là thành phần thứ j của vector \mathbf{x}_i .

Có thể thấy rằng xác suất rơi vào mỗi mặt tuân theo phân phối categorical với các tham số $\lambda_j > 0, j = 1, 2, \dots, 6$. Ta dùng $\boldsymbol{\lambda}$ để thể hiện cho cả sáu tham số này.

Với các tham số $\boldsymbol{\lambda}$, xác suất để sự kiện \mathbf{x}_i xảy ra là

$$p(\mathbf{x}_i | \boldsymbol{\lambda}) = \prod_{j=1}^6 \lambda_j^{x_i^j} \quad (4.11)$$

Khi đó, vẫn với giả sử về sự độc lập giữa các lần tung xúc sắc, ước lượng bộ tham số $\boldsymbol{\lambda}$ dựa trên việc tối đa log-likelihood ta có:

$$\boldsymbol{\lambda} = \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \left[\prod_{i=1}^N p(\mathbf{x}_i | \boldsymbol{\lambda}) \right] = \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \left[\prod_{i=1}^N \prod_{j=1}^6 \lambda_j^{x_i^j} \right] \quad (4.12)$$

$$= \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \left[\prod_{j=1}^6 \lambda_j^{\sum_{i=1}^N x_i^j} \right] = \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \left[\prod_{j=1}^6 \lambda_j^{n_j} \right] \quad (4.13)$$

$$= \underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \left[\sum_{j=1}^6 n_j \log(\lambda_j) \right] \quad (4.14)$$

Khác với bài toán (4.9) một chút, chúng ta không được quên điều kiện $\sum_{j=1}^6 \lambda_j = 1$. Ta có bài toán tối ưu có ràng buộc sau đây

$$\max_{\boldsymbol{\lambda}} \sum_{j=1}^6 n_j \log(\lambda_j) \quad \text{thỏa mãn: } \sum_{j=1}^6 \lambda_j = 1 \quad (4.15)$$

Bài toán tối ưu này có thể được giải bằng phương pháp nhân tử Lagrange (xem Phụ lục A).

Lagrangian của bài toán này là

$$\mathcal{L}(\lambda, \mu) = \sum_{j=1}^6 n_j \log(\lambda_j) + \mu(1 - \sum_{j=1}^6 \lambda_j) \quad (4.16)$$

Nghiệm của bài toán là nghiệm của hệ đạo hàm của $\mathcal{L}(\cdot)$ theo từng biến bằng 0

$$\frac{\partial \mathcal{L}(\lambda, \mu)}{\partial \lambda_j} = \frac{n_j}{\lambda_j} - \mu = 0, \quad \forall j = 1, 2, \dots, 6 \quad (4.17)$$

$$\frac{\partial \mathcal{L}(\lambda, \mu)}{\partial \mu} = 1 - \sum_{j=1}^6 \lambda_j = 0 \quad (4.18)$$

Từ (4.17) ta có $\lambda_j = \frac{n_j}{\mu}$. Thay vào (4.18),

$$\sum_{j=1}^6 \frac{n_j}{\mu} = 1 \Rightarrow \mu = \sum_{j=1}^6 n_j = N \quad (4.19)$$

Từ đó ta có ước lượng $\lambda_j = \frac{n_j}{N}$, $\forall j = 1, 2, \dots, 6$.

Qua hai ví dụ trên ta thấy MLE cho kết quả khá hợp lý.

Ví dụ 3: Univariate normal distribution

Bài toán: Khi thực hiện một phép đo, giả sử rằng rất khó để có thể đo *chính xác* độ dài của một vật. Thay vào đó, người ta thường đo vật đó nhiều lần rồi suy ra kết quả, với giả thiết rằng các phép đo là độc lập với nhau và kết quả mỗi phép đo là một phân phối chuẩn. Ước lượng chiều dài của vật đó dựa trên các kết quả đo được.

Lời giải: Vì biết rằng kết quả phép đo tuân theo phân phối chuẩn, ta sẽ cố gắng đi xây dựng phân phối chuẩn đó. Chiều dài của vật có thể được coi là giá trị mà hàm mật độ xác suất đạt giá trị cao nhất, tức khả năng rơi vào khoảng giá trị xung quanh nó là lớn nhất. Trong phân phối chuẩn, ta biết rằng hàm mật độ xác suất đạt giá trị lớn nhất tại chính kỳ vọng của phân phối đó. Chú ý rằng kỳ vọng của phân phối và kỳ vọng của dữ liệu quan sát được có thể không chính xác bằng nhau, nhưng rất gần nhau. Nếu ước lượng kỳ vọng của phân phối như cách làm dưới đây sử dụng MLE, ta sẽ thấy rằng kỳ vọng của dữ liệu chính là đánh giá tốt nhất cho kỳ vọng của phân phối.

Thật vậy, giả sử các kích thước quan sát được là x_1, x_2, \dots, x_N . Ta cần đi tìm một phân phối chuẩn, được mô tả bởi một giá trị kỳ vọng μ và phương sai σ^2 , sao cho các giá trị x_1, x_2, \dots, x_N là *likely nhất*. Ta đã biết rằng, hàm mật độ xác suất tại x_i của một phân phối chuẩn có kỳ vọng μ và phương sai σ^2 là

$$p(x_i | \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2}\right) \quad (4.20)$$

Vậy, để đánh giá μ và σ , ta sử dụng MLE với giả thiết rằng kết quả các phép đo là độc lập:

$$\mu, \sigma = \operatorname{argmax}_{\mu, \sigma} \left[\prod_{i=1}^N p(x_i | \mu, \sigma^2) \right] \quad (4.21)$$

$$= \operatorname{argmax}_{\mu, \sigma} \left[\frac{1}{(2\pi\sigma^2)^{N/2}} \exp \left(-\frac{\sum_{i=1}^N (x_i - \mu)^2}{2\sigma^2} \right) \right] \quad (4.22)$$

$$= \operatorname{argmax}_{\mu, \sigma} \left[-N \log(\sigma) - \frac{\sum_{i=1}^N (x_i - \mu)^2}{2\sigma^2} \triangleq J(\mu, \sigma) \right] \quad (4.23)$$

Ta đã lấy log của hàm bên trong dấu ngoặc vuông của (4.22) để được (4.23), phần hằng số có chứa 2π cũng đã được bỏ đi vì nó không ảnh hưởng tới kết quả.

Để tìm μ và σ , ta giải hệ phương trình đạo hàm của $J(\mu, \sigma)$ theo mỗi biến bằng không:

$$\frac{\partial J}{\partial \mu} = \frac{1}{\sigma^2} \sum_{i=1}^N (x_i - \mu) = 0 \quad (4.24)$$

$$\frac{\partial J}{\partial \sigma} = -\frac{N}{\sigma} + \frac{1}{\sigma^3} \sum_{i=1}^N (x_i - \mu)^2 = 0 \quad (4.25)$$

$$\Rightarrow \mu = \frac{\sum_{i=1}^N x_i}{N}, \quad \sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N} \quad (4.26)$$

Kết quả thu được không có gì bất ngờ.

Ví dụ 4: Multivariate normal distribution

Bài toán: Giả sử tập dữ liệu ta thu được là các giá trị nhiều chiều $\mathbf{x}_1, \dots, \mathbf{x}_N$ tuân theo phân phối chuẩn. Hãy đánh giá các tham số, vector kỳ vọng $\boldsymbol{\mu}$ và ma trận hiệp phương sai $\boldsymbol{\Sigma}$ của phân phối này dựa trên MLE, giả sử rằng các $\mathbf{x}_1, \dots, \mathbf{x}_N$ là độc lập.

Lời giải: Việc chứng minh các công thức

$$\boldsymbol{\mu} = \frac{\sum_{i=1}^N \mathbf{x}_i}{N} \quad (4.27)$$

$$\boldsymbol{\Sigma} = \frac{1}{N} \sum_{i=1}^N (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T \quad (4.28)$$

xin được dành lại cho bạn đọc như một bài tập nhỏ. Dưới đây là một vài gợi ý:

- Hàm mật độ xác suất của phân phối chuẩn nhiều chiều là

$$p(\mathbf{x} | \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2} \|\boldsymbol{\Sigma}\|^{1/2}} \exp \left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right) \quad (4.29)$$

Chú ý rằng ma trận hiệp phương sai $\boldsymbol{\Sigma}$ là xác định dương nên có nghịch đảo.

- Một vài đạo hàm theo ma trận:

$$\nabla_{\Sigma} \log |\Sigma| = (\Sigma^{-1})^T \triangleq \Sigma^{-T} \quad (\text{chuyển vị của nghịch đảo}) \quad (4.30)$$

$$\nabla_{\Sigma} (\mathbf{x}_i - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x}_i - \boldsymbol{\mu}) = -\Sigma^{-T} (\mathbf{x}_i - \boldsymbol{\mu}) (\mathbf{x}_i - \boldsymbol{\mu})^T \Sigma^{-T} \quad (4.31)$$

(Xem thêm Matrix Calculus, mục D.2.1 và D.2.4 tại <https://goo.gl/JKg631>.)

4.3 Maximum a Posteriori

4.3.1 Ý tưởng

Quay lại với ví dụ 1 về tung đồng xu. Nếu tung đồng xu 5000 lần và nhận được 1000 lần *head*, ta có thể đánh giá xác suất của *head* là 1/5 và việc đánh giá này là đáng tin vì số mẫu là lớn. Nếu tung 5 lần và chỉ nhận được 1 mặt *head*, theo MLE, xác suất để có một mặt *head* được đánh giá là 1/5. Tuy nhiên với chỉ 5 kết quả, ước lượng này là không đáng tin, nhiều khả năng việc đánh giá đã bị *overfitting*. Khi tập huấn luyện quá nhỏ (*low-training*) chúng ta cần phải quan tâm tới một vài giả thiết của các tham số. Trong ví dụ này, giả thiết của chúng ta là xác suất nhận được mặt *head* phải gần 1/2.

Maximum A Posteriori (MAP) ra đời nhằm giải quyết vấn đề này. Trong MAP, chúng ta giới thiệu một giả thiết biết trước, được gọi là *prior*, của tham số θ . Từ giả thiết này, chúng ta có thể suy ra các khoảng giá trị và phân bố của tham số.

Ngược với MLE, trong MAP, chúng ta sẽ đánh giá tham số như là một xác suất có điều kiện của dữ liệu:

$$\theta = \operatorname{argmax}_{\theta} \underbrace{p(\theta | \mathbf{x}_1, \dots, \mathbf{x}_N)}_{\text{posterior}} \quad (4.32)$$

Biểu thức $p(\theta | \mathbf{x}_1, \dots, \mathbf{x}_N)$ còn được gọi là *xác suất posterior* của θ . Chính vì vậy mà việc ước lượng θ theo (4.32) được gọi là *Maximum A Posteriori*.

Thông thường, hàm tối ưu trong (4.32) khó xác định dạng một cách trực tiếp. Chúng ta thường biết điều ngược lại, tức nếu biết tham số, ta có thể tính được hàm mật độ xác suất của dữ liệu. Vì vậy, để giải bài toán MAP, ta thường sử dụng quy tắc Bayes. Bài toán MAP thường được biến đổi thành

$$\theta = \operatorname{argmax}_{\theta} p(\theta | \mathbf{x}_1, \dots, \mathbf{x}_N) = \operatorname{argmax}_{\theta} \left[\frac{\overbrace{p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta)}^{\text{likelihood}} \overbrace{p(\theta)}^{\text{prior}}}{\underbrace{p(\mathbf{x}_1, \dots, \mathbf{x}_N)}_{\text{evidence}}} \right] \quad (4.33)$$

$$= \operatorname{argmax}_{\theta} [p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta) p(\theta)] \quad (4.34)$$

$$= \operatorname{argmax}_{\theta} \left[\prod_{i=1}^N p(\mathbf{x}_i | \theta) p(\theta) \right] \quad (4.35)$$

Đẳng thức (4.33) xảy ra theo quy tắc Bayes. Đẳng thức (4.34) xảy ra vì mẫu số của (4.33) không phụ thuộc vào tham số θ . Đẳng thức (4.35) xảy ra nếu chúng ta giả thiết về sự độc lập giữa các \mathbf{x}_i . Chú ý rằng giả thiết độc lập thường xuyên được sử dụng.

Như vậy, điểm khác biệt lớn nhất giữa hai bài toán tối ưu MLE và MAP là việc hàm mục tiêu của MAP có thêm $p(\theta)$, tức phân phối của θ . Phân phối này chính là những thông tin ta biết trước về θ và được gọi là *prior*. Ta kết luận rằng **posterior tỉ lệ thuận với tích của likelihood và prior**.

Vậy chọn *prior* thế nào? chúng ta cùng làm quen với một khái niệm mới: *conjugate prior*.

4.3.2 Conjugate prior

Nếu phân phối xác suất posterior $p(\theta|\mathbf{x}_1, \dots, \mathbf{x}_N)$ có cùng dạng (*same family*) với phân phối xác suất $p(\theta)$, prior và posterior được gọi là *conjugate distributions*, và $p(\theta)$ được gọi là *conjugate prior* cho hàm likelihood $p(\mathbf{x}_1, \dots, \mathbf{x}_N|\theta)$. Nghiệm của bài toán MAP và MLE có cấu trúc giống nhau.

Một vài cặp các *conjugate distributions*¹:

- Nếu likelihood function là một Gaussian (phân phối chuẩn), và prior cho vector kỳ vọng cũng là một Gaussian, thế thì phân phối posterior cũng là một Gaussian. Ta nói rằng Gaussian conjugate với chính nó (hay còn gọi là *self-conjugate*).
- Nếu likelihood function là một Gaussian và prior cho phương sai là một phân phối gamma², phân phối posterior cũng là một Gaussian. Ta nói rằng phân phối gamma là conjugate prior cho phương sai của Gaussian. Chú ý rằng phương sai có thể được coi là một biến giúp đo *độ chính xác* của mô hình. Phương sai càng nhỏ thì độ chính xác càng cao.
- Phân phối Beta là conjugate của phân phối Bernoulli.
- Phân phối Dirichlet là conjugate của phân phối categorical.

4.3.3 Hyperparameters

Xét một ví dụ nhỏ với phân phối Bernoulli với hàm mật độ xác suất:

$$p(x|\lambda) = \lambda^x(1 - \lambda)^{1-x} \quad (4.36)$$

và conjugate của nó, phân phối Beta, có hàm phân mật độ xác suất:

$$p(\lambda) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} \lambda^{\alpha-1}(1 - \lambda)^{\beta-1} \quad (4.37)$$

Bỏ qua thừa số hằng số chỉ mang mục đích chuẩn hoá cho tích phân của hàm mật độ xác suất bằng một, ta có thể nhận thấy rằng phần còn lại của phân phối Beta có cùng họ (*family*)

¹ Đọc thêm: *Conjugate prior*–Wikipedia (<https://goo.gl/E2SHbD>).

² *Gamma distribution*–Wikipedia, (<https://goo.gl/kdWd2R>).

với phân phối Bernoulli. Cụ thể, nếu sử dụng phân phối Beta làm *prior* cho tham số λ , và bỏ qua phần thừa số hằng số, posterior sẽ có dạng

$$\begin{aligned} p(\lambda|x) &\propto p(x|\lambda)p(\lambda) \\ &\propto \lambda^{x+\alpha-1}(1-\lambda)^{1-x+\beta-1} \end{aligned} \quad (4.38)$$

trong đó, α là ký hiệu của *tỉ lệ với*.

Nhận thấy rằng (4.38) vẫn có dạng của một phân phối Bernoulli. Chính vì vậy mà phân phối Beta được gọi là một *conjugate prior* cho phân phối Bernoulli.

Trong ví dụ này, tham số λ phụ thuộc vào hai tham số khác là α và β . Để tránh nhầm lẫn, hai tham số (α, β) được gọi là *siêu tham số* (*hyperparameters*).

Quay trở lại ví dụ về bài toán tung đồng xu N lần có n lần nhận được mặt *head* và $m = N - n$ lần nhận được mặt *tail*. Nếu sử dụng MLE, ta nhận được ước lượng $\lambda = n/M$. Nếu sử dụng MAP với prior là một Beta $[\alpha, \beta]$ thì kết quả sẽ thay đổi thế nào?

Bài toán tối ưu MAP:

$$\begin{aligned} \lambda &= \operatorname{argmax}_{\lambda} [p(x_1, \dots, x_N|\lambda)p(\lambda)] \\ &= \operatorname{argmax}_{\lambda} \left[\left(\prod_{i=1}^N \lambda^{x_i} (1-\lambda)^{1-x_i} \right) \lambda^{\alpha-1} (1-\lambda)^{\beta-1} \right] \\ &= \operatorname{argmax}_{\lambda} \left[\lambda^{\sum_{i=1}^N x_i + \alpha - 1} (1-\lambda)^{N - \sum_{i=1}^N x_i + \beta - 1} \right] \\ &= \operatorname{argmax}_{\lambda} \left[\lambda^{n+\alpha-1} (1-\lambda)^{m+\beta-1} \right] \end{aligned} \quad (4.39)$$

Bài toán tối ưu (4.39) chính là bài toán tối ưu (4.38) với tham số thay đổi một chút. Tương tự như (4.38), nghiệm của (4.39) có thể được suy ra là

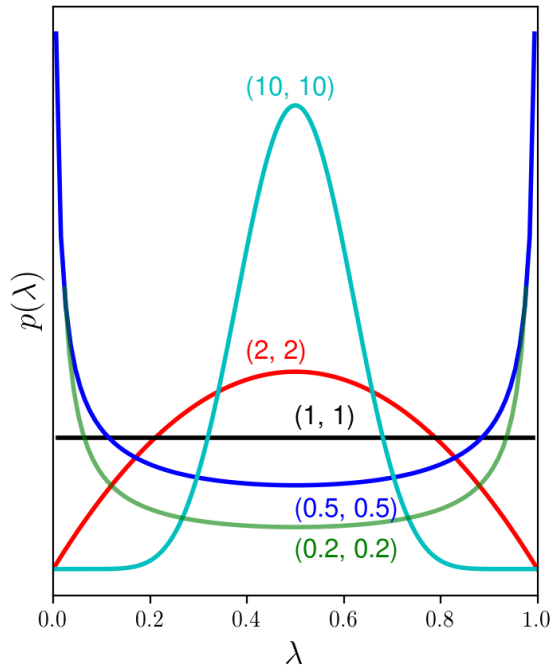
$$\lambda = \frac{n + \alpha - 1}{N + \alpha + \beta - 2} \quad (4.40)$$

Nhờ việc chọn prior phù hợp, ở đây là conjugate prior, posterior và likelihood có dạng giống nhau, khiến cho việc tối ưu bài toán MAP được thuận lợi.

Việc còn lại là chọn cặp *hyperparameters* α và β .

Chúng ta cùng xem lại hình dạng của phân phối Beta và nhận thấy rằng khi $\alpha = \beta > 1$, hàm mật độ xác suất của phân phối Beta đối xứng qua điểm 0.5 và đạt giá trị cao nhất tại 0.5. Xét Hình 4.1, ta nhận thấy rằng khi $\alpha = \beta > 1$, mật độ xác suất xung quanh điểm 0.5 nhận giá trị cao, điều này chứng tỏ λ có xu hướng gần với 0.5.

Nếu ta chọn $\alpha = \beta = 1$, ta nhận được phân phối đều vì đồ thị hàm mật độ xác suất là một đường thẳng. Lúc này, xác suất của λ tại mọi vị trí trong khoảng $[0, 1]$ là như nhau. Thực



Hình 4.1: Đồ thị hàm mật độ xác suất của phân phối Beta khi $\alpha = \beta$ và nhận các giá trị khác nhau. Khi cả hai giá trị này lớn, xác suất để λ gần 0.5 sẽ cao hơn.

chất, nếu ta thay $\alpha = \beta = 1$ vào (4.40) ta sẽ thu được $\lambda = n/N$, đây chính là ước lượng thu được bằng MLE. MLE là một trường hợp đặc biệt của MAP khi prior là một phân phối đều.

Nếu ta chọn $\alpha = \beta = 2$, ta sẽ thu được: $\lambda = \frac{n+1}{N+2}$. Chẳng hạn khi $N = 5, n = 1$ như trong ví dụ. MLE cho kết quả $\lambda = 1/5$, MAP sẽ cho kết quả $\lambda = 2/7$, gần với $1/2$ hơn.

Nếu chọn $\alpha = \beta = 10$ ta sẽ có $\lambda = (1+9)/(5+18) = 10/23$. Ta thấy rằng khi $\alpha = \beta$ và càng lớn thì ta sẽ thu được λ càng gần $1/2$. Điều này có thể dễ nhận thấy vì prior nhận giá trị rất cao tại 0.5 khi các siêu tham số $\alpha = \beta$ lớn.

4.3.4 MAP giúp tránh overfitting

Việc chọn các hyperparameter thường được dựa trên thực nghiệm, chẳng hạn bằng cross-validation. Việc thử nhiều bộ tham số rồi chọn ra bộ tốt nhất là việc mà các kỹ sư machine learning thường xuyên phải đối mặt. Cũng giống như việc chọn regularization parameter để tránh overfitting vậy.

Nếu viết lại bài toán MAP dưới dạng:

$$\theta = \underset{\theta}{\operatorname{argmax}} p(\mathbf{X}|\theta)p(\theta) \quad (4.41)$$

$$= \underset{\lambda}{\operatorname{argmax}} \left[\underbrace{\log p(\mathbf{X}|\theta)}_{\text{likelihood}} + \log \underbrace{p(\theta)}_{\text{prior}} \right] \quad (4.42)$$

ta có thể thấy rằng hàm mục tiêu có dạng $\mathcal{L}(\theta) + \lambda R(\theta)$ giống như trong regularization, với hàm log-likelihood đóng vai trò như hàm mất mát $\mathcal{L}(\theta)$, và log của prior đóng vai trò như hàm $R(\theta)$. Ta có thể nói rằng, MAP chính là một phương pháp giúp tránh overfitting trong các mô hình machine learning thống kê. MAP đặc biệt hữu ích khi tập huấn luyện là nhỏ.

4.4 Tóm tắt

- Khi sử dụng các mô hình thống kê machine learning, chúng ta thường xuyên phải ước lượng các tham số của mô hình θ , đại diện cho các tham số của các phân phối xác suất. Có hai phương pháp phổ biến được sử dụng để ước lượng θ là Maximum Likelihood Estimation (MLE) và Maximum A Posterior Estimation (MAP).
- Với MLE, việc xác định tham số θ được thực hiện bằng cách đi tìm các tham số sao cho xác suất của tập huấn luyện, hay còn gọi là *likelihood*, là lớn nhất:

$$\theta = \underset{\theta}{\operatorname{argmax}} p(\mathbf{x}_1, \dots, \mathbf{x}_N | \theta) \quad (4.43)$$

- Để giải bài toán tối ưu này, giả thiết các dữ liệu \mathbf{x}_i độc lập thường được sử dụng. Và bài toán MLP trở thành:

$$\theta = \underset{\theta}{\operatorname{argmax}} \prod_{i=1}^N p(\mathbf{x}_i | \theta) \quad (4.44)$$

- Với MAP, các tham số được đánh giá bằng cách tối đa *posterior*:

$$\theta = \underset{\theta}{\operatorname{argmax}} p(\theta | \mathbf{x}_1, \dots, \mathbf{x}_N) \quad (4.45)$$

- Quy tắc Bayes và giả thiết về sự độc lập của dữ liệu thường được sử dụng:

$$\theta = \underset{\theta}{\operatorname{argmax}} \left[\prod_{i=1}^N p(\mathbf{x}_i | \theta) p(\theta) \right] \quad (4.46)$$

Hàm mục tiêu ở đây chính là tích của *likelihood* và *prior*.

- *Prior* thường được chọn dựa trên các thông tin biết trước của tham số, và phân phối được chọn thường là các *conjugate distribution* với likelihood, tức các phân phối khiến việc nhân thêm *prior* vẫn giữ được cấu trúc giống như *likelihood*.
- MAP có thể được coi là một phương pháp giúp tránh overfitting. MAP thường mang lại hiệu quả cao hơn MLE với trường hợp có ít dữ liệu huấn luyện.

Tổng quan về machine learning

Các khái niệm cơ bản

Nội dung của chương này được tham khảo chủ yếu từ Mục 5.1 trong cuốn sách *Deep learning* (Goodfellow, 2016).

Một thuật toán machine learning là một thuật toán có khả năng *học tập* từ dữ liệu. Vậy thực sự *học tập* ở đây có nghĩa như thế nào? Theo Mitchell trong cuốn *Machine Learning* [M⁺97], Mục 1.1, thì “A computer program is said to **learn** from *experience E* with respect to some *tasks T* and *performance measure P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*.”

Tạm dịch:

Định nghĩa 5.1: Học (chương trình máy tính)

Một chương trình máy tính được gọi là **học** từ *kinh nghiệm E* để hoàn thành *nhiệm vụ T*, với hiệu quả được đo bằng *phép đánh giá P*, nếu hiệu quả của nó khi thực hiện nhiệm vụ *T*, khi được đánh giá bởi *P*, cải thiện theo *kinh nghiệm E*.

Trong chương này, chúng ta sẽ đi vào từng khái niệm *task*, *performance measure*, và *experience* thông qua các ví dụ.

5.1 Nhiệm vụ, T

Các *nhiệm vụ* trong machine learning thường được mô tả thông qua việc một hệ thống machine learning xử lý một *điểm dữ liệu (data point)* như thế nào. Trong bài toán phân loại ảnh, mỗi ảnh là một điểm dữ liệu. Trong bài toán phân nhóm khách hàng, mỗi khách hàng là một điểm dữ liệu. Trong bài toán xác định một tin nhắn có là rác hay không, mỗi tin nhắn là một điểm dữ liệu. Mỗi điểm dữ liệu bao gồm nhiều *đặc trưng (feature)* khác nhau, mỗi feature thường được biểu diễn dưới dạng một con số. Chúng ta thường biểu diễn một

điểm dữ liệu như một vector¹ $\mathbf{x} \in \mathbb{R}^d$ trong đó mỗi phần tử x_i là một đặc trưng, vector này thường được gọi là *vector đặc trưng* (*feature vector*). Ví dụ, trong một bức ảnh, mỗi giá trị của một điểm ảnh có thể coi là một đặc trưng, vector chứa toàn bộ giá trị các pixel của ảnh có thể coi là một vector đặc trưng. Chương 6 sẽ bàn sâu thêm về vector đặc trưng của dữ liệu.

Rất nhiều *nhiệm vụ* phức tạp có thể được giải quyết bằng machine learning. Dưới đây là một trong những bài toán phổ biến nhất của machine learning.

5.1.1 Classification

Classification, hay *phân loại*, *phân lớp*. Đây là một trong những bài toán được nghiên cứu nhiều nhất trong machine learning. Trong bài toán này, chương trình sẽ được yêu cầu chỉ ra *nhãn*, hay *lớp* (*label*) của một điểm dữ liệu. Nhãn này thường là một phần tử trong một tập hợp có C phần tử khác nhau. Mỗi phần tử trong tập hợp này được gọi là một *lớp* (*class*), và thường được đánh số từ 1 đến C . Để giải bài toán này, ta thường phải xây dựng một hàm số $f: \mathbb{R}^d \rightarrow \{1, 2, \dots, C\}$. Khi $y = f(\mathbf{x})$, mô hình gán cho một điểm dữ liệu được mô tả bởi vector đặc trưng \mathbf{x} một nhãn được xác định bởi số y .

Ví dụ: trong nhận dạng chữ số viết tay, ta có ảnh của hàng nghìn ví dụ của mỗi chữ số được viết bởi nhiều người khác nhau. Các bức ảnh này cùng với nhãn của chúng được đưa vào một thuật toán machine learning. Sau khi thuật toán này *học* được một mô hình, tức một hàm số mà đầu vào là một bức ảnh và đầu ra là một chữ số, khi nhận được một bức ảnh mới mà mô hình **chưa nhìn thấy bao giờ**, nó sẽ dự đoán bức ảnh đó chứa chữ số nào. Ví dụ này khá giống với cách học của con người khi còn nhỏ. Ta đưa bảng chữ cái cho một đứa trẻ và chỉ cho chúng đây là chữ A, đây là chữ B. Sau một vài lần được dạy thì trẻ có thể nhận biết được đâu là chữ A, đâu là chữ B mà chúng chưa nhìn thấy bao giờ.

Có một biến thể nhỏ ở đầu ra của hàm số $f(\mathbf{x})$ khi đầu ra không phải là một số mà là một vector $\mathbf{y} \in \mathbb{R}^C$ trong đó y_c chỉ ra xác suất để điểm dữ liệu \mathbf{x} rơi vào lớp thứ c . Lớp được chọn cuối cùng là lớp có xác suất rơi vào là cao nhất. Việc sử dụng xác suất này đôi khi rất quan trọng, nó giúp chỉ ra *độ chắc chắn* (*confidence*) của mô hình. Nếu xác suất cao nhất là cao hơn nhiều so với các xác suất còn lại, ta nói mô hình có độ chắc chắn là cao khi phân lớp điểm dữ liệu \mathbf{x} . Ngược lại, nếu độ chênh lệch giữa xác suất cao nhất và các xác suất tiếp theo là nhỏ, thì khả năng mô hình đã phân loại nhầm là cao hơn.

5.1.2 Regression

Nếu nhãn không được chia thành các nhóm mà là các giá trị thực (có thể vô hạn) thì bài toán được gọi là *hồi quy*, một số tài liệu gọi là *tiên lượng* (*regression*). Trong bài toán này, ta cần xây dựng một hàm số $f: \mathbb{R}^d \rightarrow \mathbb{R}$.

¹ Có những loại dữ liệu không được biểu diễn dưới dạng một vector mà có thể là một ma trận—khi giữ nguyên một bức ảnh trong không gian hai chiều, hoặc một *tensor*—mảng nhiều chiều—khi xem các bức ảnh với nhiều channel khác nhau. Trong cuốn sách này, chúng ta chỉ xét các điểm dữ liệu dưới dạng vector, hoặc *vector hoá* (*vectorization*) các điểm dữ liệu nhiều chiều.

Ví dụ 1: Ước lượng một căn nhà rộng x m², có y phòng ngủ và cách trung tâm thành phố z km sẽ có giá khoảng bao nhiêu?

Ví dụ 2: Microsoft có một ứng dụng dự đoán giới tính và tuổi dựa trên khuôn mặt (<http://how-old.net/>). Phần dự đoán giới tính có thể được coi là một thuật toán classification, phần dự đoán tuổi có thể coi là một thuật toán regression. Chú ý rằng phần dự đoán tuổi cũng có thể coi là classification nếu ta coi tuổi là một số nguyên dương không lớn hơn 150, chúng ta sẽ có 150 class (lớp) khác nhau.

Bài toán regression có thể mở rộng ra việc dự đoán nhiều đầu ra cùng một lúc, khi đó, hàm cần tìm sẽ là $f : \mathbb{R}^d \rightarrow \mathbb{R}^m$. Một ví dụ là bài toán *single image super resolution*, ở đó, hệ thống cần tạo ra một bức ảnh có độ phân giải cao dựa trên một ảnh có độ phân giải thấp hơn. Khi đó, việc dự đoán giá trị của các pixel trong ảnh đầu ra là một bài toán regression với nhiều đầu ra.

5.1.3 Machine translation

Trong bài toán này, đầu vào là một câu, đoạn, hay bài văn trong một ngôn ngữ, và chương trình máy tính được yêu cầu chuyển đổi nó sang một ngôn ngữ khác. Lời giải cho bài toán này gần đây đã có nhiều bước phát triển vượt bậc dựa trên các thuật toán deep learning.

5.1.4 Clustering

Clustering là bài toán *phân nhóm* toàn bộ dữ liệu \mathcal{X} thành các nhóm nhỏ dựa trên sự liên quan giữa các dữ liệu trong mỗi nhóm.

Ví dụ: phân nhóm khách hàng dựa trên hành vi mua hàng. Điều này cũng giống như việc ta đưa cho một đứa trẻ rất nhiều mảnh ghép với các hình thù và màu sắc khác nhau, ví dụ tam giác, vuông, tròn với màu xanh và đỏ, sau đó yêu cầu trẻ phân chúng thành từng nhóm. Mặc dù không cho trẻ biết mảnh nào tương ứng với hình nào hoặc màu nào, nhiều khả năng chúng vẫn có thể phân loại các mảnh ghép theo màu hoặc hình dạng.

5.1.5 Completion

Completion là bài toán *điền* những giá trị còn thiếu của một điểm dữ liệu. Trong nhiều bài toán thực tế, việc thu thập toàn bộ thông tin của một điểm dữ liệu, ví dụ khách hàng, là không khả thi. Nhiệm vụ của bài toán này là dựa trên mối tương quan giữa các điểm dữ liệu để dự đoán những giá trị còn thiếu. *Các hệ thống khuyến nghị (recommendation system)* là một ví dụ điển hình của loại này.

Bạn đọc có thể đọc thêm về các bài toán *xếp hạng (ranking)*, *thu thập thông tin (information retrieval)*, *giảm nhiễu (denoising)*, v.v..

5.2 Phép đánh giá, P

Để kiểm tra năng lực của một thuật toán machine learning, chúng ta cần phải thiết kế các phép đánh giá có thể đo đạc được kết quả.

Thông thường, khi thực hiện một thuật toán machine learning, dữ liệu sẽ được chia thành hai phần riêng biệt: *tập huấn luyện (training set)* và *tập kiểm thử (test set)*. Tập huấn luyện sẽ được dùng để tìm các tham số mô hình. Tập kiểm thử được dùng để đánh giá năng lực của mô hình tìm được. Có một điểm cần lưu ý rằng khi tìm các tham số mô hình, ta chỉ được dùng các thông tin trong tập huấn luyện. Việc đánh giá có thể được áp dụng lên cả hai tập hợp. Muốn mô hình thực hiện tốt trên tập kiểm thử thì nó trước hết phải hoạt động tốt trên tập huấn luyện.

Lưu ý: Ranh giới giữa tập huấn luyện và tập kiểm thử đôi khi không rõ ràng. Các thuật toán thực tế liên tục được cập nhật dựa trên dữ liệu mới thêm vào, các thuật toán này được gọi là *online learning* hoặc *online training*. Phần dữ liệu mới này ban đầu không được hệ thống sử dụng để xây dựng mô hình, nhưng về sau có thể được mô hình sử dụng để cải tiến. Ngược với *online learning* là *offline learning*, ở đó hệ thống xây dựng mô hình *một lần* dựa trên một tập chính là tập huấn luyện. Các điểm dữ liệu không được dùng trong quá trình xây dựng hệ thống được coi là tập kiểm thử. Trong cuốn sách này, khi không đề cập gì thêm, các thuật toán được ngầm hiểu là *offline learning*, trong đó *training set* là tập hợp được dùng để xây dựng mô hình ban đầu, *test set* là tập hợp được dùng để đánh giá hiệu quả của mô hình được xây dựng đó.

5.3 Kinh nghiệm, E

Việc huấn luyện các mô hình machine learning có thể coi là việc cho chúng *trải nghiệm* trên các *tập dữ liệu (dataset)*—chính là *training set*. Các tập dữ liệu khác nhau sẽ cho các mô hình các trải nghiệm khác nhau. Chất lượng của các tập dữ liệu này cũng ảnh hưởng tới hiệu năng của mô hình.

Dựa trên tính chất của các tập dữ liệu, các thuật toán machine learning có thể phân loại thành hai nhóm chính là *học không giám sát (unsupervised learning)* và *học có giám sát (supervised learning)*.

Supervised learning là thuật toán dự đoán đầu ra của một hoặc nhiều dữ liệu mới dựa trên các cặp (*đầu vào, đầu ra*) đã biết từ trước. Supervised learning là nhóm phổ biến nhất trong các thuật toán machine learning.

Một cách toán học, supervised learning là khi chúng ta có một tập hợp biến đầu vào $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ và một tập hợp đầu ra tương ứng $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_N\}$, trong đó $\mathbf{x}_i, \mathbf{y}_i$ là các vector. Các cặp dữ liệu biết trước $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y}$ tạo nên tập huấn luyện. Từ tập huấn luyện này, chúng ta cần tạo ra một hàm số ánh xạ mỗi phần tử từ tập \mathcal{X} sang một phần tử (xấp xỉ) tương ứng của tập \mathcal{Y} :

$$\mathbf{y}_i \approx f(\mathbf{x}_i), \quad \forall i = 1, 2, \dots, N$$

Mục đích là xấp xỉ hàm số f thật tốt để khi có một dữ liệu \mathbf{x} mới, chúng ta có thể tính được nhân tương ứng của nó $\mathbf{y} = f(\mathbf{x})$.

Ngược lại, trong **unsupervised learning**, chúng ta không biết được kết quả đầu ra mà chỉ biết các vector đặc trưng của dữ liệu đầu vào. Các thuật toán unsupervised learning sẽ dựa vào cấu trúc của dữ liệu để thực hiện một công việc nào đó, ví dụ như phân nhóm hoặc *giảm số chiều của dữ liệu (dimensionality reduction)*. Một cách toán học, unsupervised learning là khi chúng ta chỉ có dữ liệu đầu vào \mathcal{X} mà không biết đầu ra \mathcal{Y} tương ứng.

Không giống như trong supervised learning, chúng ta không biết câu trả lời chính xác cho mỗi dữ liệu đầu vào trong unsupervised learning. Giống như khi ta học, ta chỉ được đưa cho một chữ cái mà không nói đó là chữ A hay chữ B. Cụm từ *không giám sát*, hay *không ai chỉ bảo (unsupervised)* được đặt tên theo nghĩa này.

Từ góc độ xác suất thống kê, unsupervised learning trải nghiệm qua rất nhiều ví dụ (các điểm dữ liệu) \mathbf{x} và cố gắng học phân phối xác suất $p(\mathbf{x})$, hoặc các tính chất của phân phối của dữ liệu một cách trực tiếp hoặc gián tiếp. Trong khi đó, supervised learning quan sát các ví dụ \mathbf{x} và các kết quả tương ứng \mathbf{y} , sau đó cố gắng học cách dự đoán \mathbf{y} từ \mathbf{x} thông qua việc đánh giá xác suất có điều kiện $p(\mathbf{y}|\mathbf{x})$. Xác suất này có thể diễn đạt bằng lời là biết rằng một điểm dữ liệu có vector đặc trưng là \mathbf{x} , xác suất để đầu ra của nó bằng \mathbf{y} là bao nhiêu.

Ranh giới giữa unsupervised learning và supervised learning đôi khi là không rõ ràng. Thông thường, người ta thường coi các bài classification, regression là supervised learning, các bài clustering hay *density estimation (ước lượng một phân phối)* là unsupervised learning.

Có những bài toán mà dữ liệu được dùng để huấn luyện bao gồm cả những dữ liệu có nhãn và chưa được gán nhãn. Các bài toán khi chúng ta có một lượng lớn dữ liệu \mathcal{X} nhưng chỉ một phần trong chúng được gán nhãn được gọi là *học bán giám sát*, hay **semi-supervised learning**. Những bài toán thuộc nhóm này nằm giữa hai nhóm được nêu bên trên.

Một ví dụ điển hình của nhóm này là chỉ có một phần ảnh hoặc văn bản được gán nhãn (ví dụ bức ảnh về người, động vật hoặc các văn bản khoa học, chính trị) và phần lớn các bức ảnh/văn bản khác chưa được gán nhãn được thu thập từ internet. Thực tế cho thấy rất nhiều các bài toán machine learning thuộc vào nhóm này vì việc thu thập dữ liệu có nhãn tốn rất nhiều thời gian và có chi phí cao. Rất nhiều loại dữ liệu, ví dụ như ảnh y học, thậm chí cần phải có chuyên gia mới gán nhãn được. Ngược lại, dữ liệu chưa có nhãn có thể được thu thập với chi phí thấp từ internet.

Có những thuật toán machine learning không luôn trải nghiệm trên một tập dữ liệu cố định. Ví dụ, *học củng cố (reinforcement learning)* trải nghiệm trực tiếp với môi trường xung quanh, liên tục nhận phản hồi từ môi trường để tự cải thiện hành vi của hệ thống trong các môi trường mới. Các ví dụ điển hình của reinforcement learning là việc huấn luyện cho xe tự lái dựa vào ảnh nhận từ camera và điều khiển tay lái cũng như tốc độ của xe. Reinforcement learning hiện nay chủ yếu được áp dụng vào các trò chơi, khi mà máy tính có thể mô phỏng được các trạng thái của môi trường và huấn luyện thuật toán thông qua rất nhiều vòng lặp.

Ví dụ 1: AlphaGo gần đây nổi tiếng với việc chơi cờ vây thắng cả con người (<https://goo.gl/PzKcvP>). Cờ vây được xem là có độ phức tạp cực kỳ cao² với tổng số nước đi là xấp xỉ 10^{761} , so với cờ vua là 10^{120} và tổng số nguyên tử trong toàn vũ trụ là khoảng 10^{80} !! Hệ thống phải chọn ra một *đường đi nước bước* tối ưu trong số hàng nghìn tỉ tỉ lựa chọn, và tất nhiên, việc thử tất cả các lựa chọn là không khả thi. Về cơ bản, AlphaGo bao gồm các thuật toán thuộc cả supervised learning và reinforcement learning. Trong phần supervised learning, dữ liệu từ các ván cờ do con người chơi với nhau được đưa vào để huấn luyện. Tuy nhiên, mục đích cuối cùng của AlphaGo không phải là chơi như con người mà phải thậm chí thắng cả con người. Vì vậy, sau khi học xong các ván cờ của con người, AlphaGo tự chơi với chính nó với hàng triệu ván chơi để tìm ra các nước đi mới tối ưu hơn. Thuật toán trong phần tự chơi này được xếp vào loại reinforcement learning.

Gần đây, Google DeepMind đã tiến thêm một bước đáng kể với AlphaGo Zero. Hệ thống này thậm chí không cần học từ các ván cờ của con người. Nó có thể tự chơi với chính mình để tìm ra các chiến thuật tối ưu. Sau 40 ngày được huấn luyện, nó đã thắng tất cả các con người và hệ thống khác, bao gồm AlphaGo³.

Ví dụ 2: Huấn luyện cho máy tính chơi game Mario⁴. Đây là một chương trình thú vị dạy máy tính chơi game Mario. Game này đơn giản hơn cờ vây vì tại một thời điểm, người chơi chỉ phải bấm một số lượng nhỏ các nút (di chuyển, nhảy, bắn đạn) hoặc không cần bấm nút nào. Đồng thời, phản ứng của máy cũng đơn giản hơn và lặp lại ở mỗi lần chơi (tại thời điểm cụ thể sẽ xuất hiện một chướng ngại vật cố định ở một vị trí cố định). Đầu vào của thuật toán là sơ đồ của màn hình tại thời điểm hiện tại, nhiệm vụ của thuật toán là với đầu vào đó, tổ hợp phím nào nên được bấm. Việc huấn luyện này được dựa trên điểm số cho việc di chuyển được bao xa trong thời gian bao lâu trong game, càng xa và càng nhanh thì được điểm thưởng càng cao (điểm thưởng này không phải là điểm của trò chơi mà là điểm do chính người lập trình tạo ra). Thông qua huấn luyện, thuật toán sẽ tìm ra một cách tối đa số điểm trên, qua đó đạt được mục đích cuối cùng là cứu công chúa.

Reinforcement learning là một lĩnh vực thú vị trong machine learning. Rất tiếc, reinforcement learning nằm ngoài phạm vi của cuốn sách này.

5.4 Hàm mất mát và tham số mô hình

Mỗi mô hình machine learning được mô tả bởi *các tham số mô hình (model parameters)*. Công việc của một thuật toán machine learning là đi tìm các tham số mô hình phù hợp với mỗi bài toán. Việc đi tìm các tham số mô hình có liên quan mật thiết đến các phép đánh giá. Mục đích của chúng ta là đi tìm các tham số mô hình sao cho các phép đánh giá cho kết quả tốt nhất. Trong bài toán classification, kết quả tốt có thể được hiểu là ít điểm dữ liệu bị phân lớp sai nhất. Trong bài toán regression, kết quả tốt là khi sự sai lệch giữa đầu ra dự đoán và đầu ra thực sự là ít nhất.

² Google DeepMind's AlphaGo: How it works (<https://goo.gl/nDNcCy>).

³ AlphaGo Zero: Learning from scratch (<https://goo.gl/xtDjoF>).

⁴ MarI/O - Machine Learning for Video Games (<https://goo.gl/QekkRz>)

Quan hệ giữa một phép đánh giá và các tham số mô hình thường được mô tả thông qua một hàm số được gọi là *hàm mất mát* (*loss function*, hay *cost function*). Hàm mất mát này thường có giá trị nhỏ khi phép đánh giá cho kết quả tốt và ngược lại. Việc đi tìm các tham số mô hình sao cho phép đánh giá trả về kết quả tốt tương đương với việc tối thiểu hàm mất mát. Như vậy, việc xây dựng một mô hình machine learning chính là việc đi giải một bài toán tối ưu. Quá trình đó có thể được coi là quá trình *learning* của *machine*.

Tập hợp các tham số mô hình thường được ký hiệu bằng θ , hàm mất mát của mô hình thường được ký hiệu là $\mathcal{L}(\theta)$ hoặc $J(\theta)$. Bài toán tối thiểu hàm mất mát để tìm tham số mô hình thường được viết dưới dạng:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \mathcal{L}(\theta) \quad (5.1)$$

ký hiệu $\underset{\theta}{\operatorname{argmin}} \mathcal{L}(\theta)$ được hiểu là giá trị của θ để hàm số $\mathcal{L}(\theta)$ đạt giá trị nhỏ nhất. Khi sử dụng $\underset{\theta}{\operatorname{argmin}}$, chúng ta phải chỉ rõ nó được thực hiện theo các biến số nào bằng cách ghi các biến số ở dưới min (ở đây là θ). Nếu hàm số chỉ có một biến số, ta có thể bỏ qua biến số đó dưới min . Tuy nhiên, biến số nên được ghi rõ ràng để giảm thiểu sự nhầm lẫn. argmax cũng được sử dụng một cách tương tự khi ta cần tìm giá trị của các biến số để một hàm số đạt giá trị lớn nhất.

Một hàm số $\mathcal{L}(\theta)$ bất kỳ có thể có rất nhiều giá trị của θ để nó đạt giá trị nhỏ nhất, hoặc cũng có thể nó không chặn dưới. Thậm chí, việc tìm giá trị nhỏ nhất của một hàm số đôi khi là không khả thi. Trong machine learning cũng như nhiều bài toán tối ưu thực tế, việc chỉ cần tìm ra một bộ tham số θ làm cho hàm mất mát đạt giá trị nhỏ nhất, hoặc thậm chí đạt một giá trị cực tiểu⁵, thường mang lại các kết quả khả quan.

Để hiểu rõ bản chất của các thuật toán machine learning, việc nắm vững các kỹ thuật tối ưu cơ bản là rất quan trọng. Cuốn sách này có nhiều chương cung cấp các kiến thức cần thiết cho tối ưu, bao gồm tối ưu không ràng buộc (Chương 12) và tối ưu có ràng buộc (xem Phần VII).

Trong các chương tiếp theo của cuốn sách này, chúng ta sẽ dần làm quen với các thành phần cơ bản của một hệ thống machine learning.

⁵ Lưu ý rằng cực tiểu trong toán học không có nghĩa là nhỏ nhất.

Giới thiệu về feature engineering

6.1 Giới thiệu

Mỗi điểm dữ liệu trong các bài toán machine learning thường được biểu diễn bằng một vector được gọi là *vector đặc trưng* (*feature vector*)¹. Hơn nữa, trong cùng một bài toán, các feature vector của tất cả các điểm thường có kích thước như nhau. Điều này là cần thiết vì các phép toán trong mô hình (cộng, nhân ma trận, vector) yêu cầu đầu vào có cùng kích thước. Khi đó, *toàn bộ* dữ liệu có thể được lưu trong một ma trận mà mỗi hàng hoặc mỗi cột là feature vector của một điểm dữ liệu. Tuy nhiên, trên thực tế, dữ liệu thường ở dạng *thô* (*raw data*) với kích thước khác nhau. Hoặc thậm chí khi kích thước của các điểm là như nhau, việc lựa chọn, tính toán đặc trưng nào phù hợp cho mỗi bài toán là nhiệm vụ quan trọng trước tiên cần được giải quyết.

Với các bài toán *thị giác máy tính* (*computer vision*), các bức ảnh thường là các ma trận hoặc *tensor* với kích thước khác nhau. Trong bài toán nhận dạng vật thể trong ảnh, đôi khi ta cần làm thêm một bước nữa là *xác định vị trí vật thể* (*object detection*), tức là tìm các khung chứa vật thể cần dự đoán. Ví dụ, trong bài toán nhận dạng khuôn mặt, ta cần tìm được vị trí các khuôn mặt trong ảnh và cắt ra các khuôn mặt đó trước khi làm các bước tiếp theo. Ngay cả khi đã xác định được các khung chứa các khuôn mặt, ta vẫn phải làm rất nhiều việc vì hình ảnh của khuôn mặt còn phụ thuộc vào góc chụp, ánh sáng, v.v. và rất nhiều yếu tố khác nữa.

Các bài toán *xử lý ngôn ngữ tự nhiên* (*natural language processing-NLP*) cũng có khó khăn tương tự khi độ dài của các văn bản là khác nhau, thậm chí có những từ rất hiếm gặp hoặc không có trong từ điển. Cũng có khi thêm một vài từ vào văn bản mà nội dung của văn bản không đổi hoặc hoàn toàn mang nghĩa ngược lại. Hoặc cùng là một câu nói nhưng tốc độ, âm giọng của mỗi người là khác nhau, tại các thời điểm khác nhau là khác nhau.

¹ Trong các hệ thống deep learning, một bức ảnh hai chiều có thể được trực tiếp đưa vào hệ thống mà không cần qua nhiều bước feature engineering. Cuốn sách này chỉ làm việc với các đặc trưng ở dạng vector cột.

Khi làm việc với các bài toán machine learning thực tế, nhìn chung chúng ta chỉ có được dữ liệu thô chưa qua chỉnh sửa, chọn lọc. Ngoài ra, chúng ta có thể phải tìm cách loại ra những dữ liệu nhiễu, và để đưa dữ liệu thô với kích thước, hay số chiều khác nhau về cùng một chuẩn (cùng là các vector hoặc ma trận). Dữ liệu chuẩn mới này phải đảm bảo giữ được những thông tin đặc trưng cho dữ liệu thô ban đầu. Không những thế, tùy vào từng bài toán, ta cần thiết kế những phép biến đổi để có những đặc trưng phù hợp. Quá trình quan trọng này được gọi là *trích chọn đặc trưng* (*feature engineering* hay *feature extraction*).

Xin trích một câu nói (xin không dịch) của Andrew Ng²:

Coming up with features is difficult, time-consuming, requires expert knowledge. “Applied machine learning” is basically feature engineering.

Để có cái nhìn tổng quan, trong mục tiếp theo, bước feature engineering này sẽ được đặt trong một bức tranh lớn hơn.

6.2 Mô hình chung cho các bài toán Machine Learning

Phần lớn các mô hình machine learning có thể được minh họa trong Hình 6.1. Có hai bước (*phase*) lớn trong mỗi bài toán machine learning là bước huấn luyện (*training phase*) và bước kiểm thử (*test phase*). Bước huấn luyện sẽ chỉ dùng dữ liệu huấn luyện, bước kiểm thử sẽ chỉ dùng dữ liệu trong tập kiểm thử³.

6.2.1 Training phase

Có hai khối có nền màu lục cần được thiết kế:

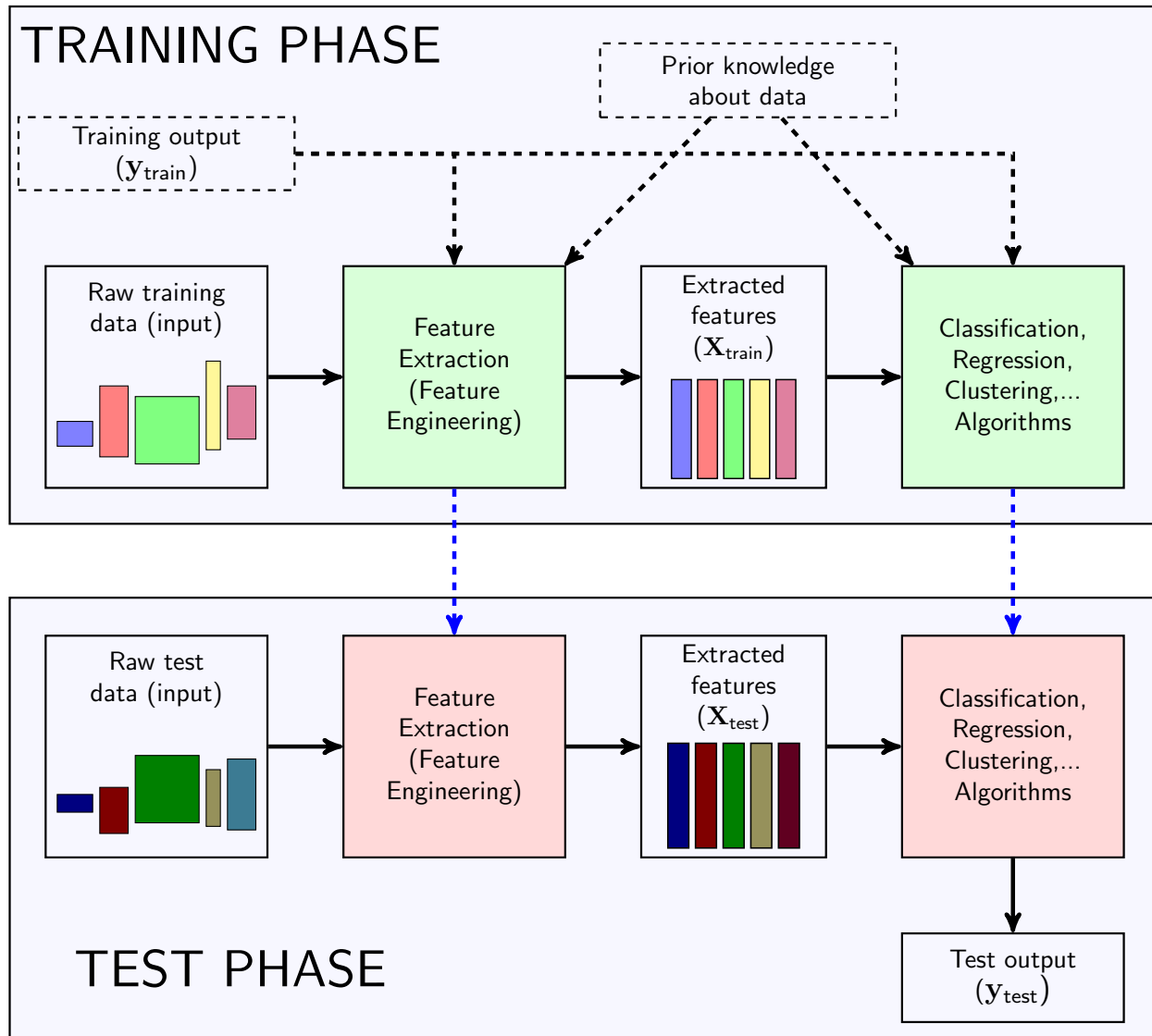
Khối thứ nhất, ***Feature Extraction***, có nhiệm vụ tạo ra một vector đặc trưng cho mỗi điểm dữ liệu đầu vào. Vector đặc trưng này thường có kích thước như nhau, bất kể dữ liệu đầu vào có kích thước như thế nào.

Đầu vào của khối Feature Extraction có thể là các yếu tố sau:

- *Dữ liệu thô ban đầu (raw training input)*. Dữ liệu thô bao gồm tất cả các thông tin ta biết về dữ liệu. Ví dụ: dữ liệu thô của một ảnh là giá trị của từng pixel; của một văn bản là từng từ, từng câu; của một file âm thanh là một đoạn tín hiệu; với bài toán dự báo thời tiết, dữ liệu thô là thông tin về hướng gió, nhiệt độ, độ ẩm, v.v.. Dữ liệu thô này thường không ở dạng vector và không có số chiều như nhau. Thậm chí có thể có số chiều như nhau nhưng số chiều quá lớn, chẳng hạn một bức ảnh màu 1000×1000 pixel sẽ có số pixel là đã là 3×10^6 (ảnh màu thường có ba channel: red, green, blue-*RGB*). Đây là một con số quá lớn, không lợi cho lưu trữ và tính toán.

² *Feature Engineering*– Wikipedia (<https://goo.gl/v4e21T>)

³ Trước khi đánh giá một mô hình trên tập kiểm thử, ta cần đảm bảo rằng mô hình đó đã làm việc tốt trên tập huấn luyện.



Hình 6.1: Mô hình thường gặp trong các bài toán machine learning.

- **output của training set.** Trong các bài toán unsupervised learning, ta không biết *output* nên hiển nhiên sẽ không có giá trị này. Trong các bài toán supervised learning, có khi dữ liệu này cũng không được sử dụng. Ví dụ, nếu *raw input* đã có cùng số chiều rồi nhưng số chiều quá lớn, ta muốn giảm số chiều của nó thì cách đơn giản nhất là *chiếu* vector đó xuống một không gian có số chiều nhỏ hơn bằng cách lấy một ma trận ngẫu nhiên nhân với nó vào bên trái. Ma trận này thường là ma trận *béo*, tức có số hàng ít hơn số cột, để đảm bảo số chiều thu được nhỏ hơn số chiều ban đầu. Việc làm này mặc dù làm mất đi thông tin, trong nhiều trường hợp vẫn mang lại hiệu quả vì đã giảm được lượng tính toán ở phần sau. Đôi khi *ma trận chiếu* không phải là ngẫu nhiên mà có thể được *học* dựa trên toàn bộ dữ liệu thô ban đầu. Trong nhiều trường hợp khác, dữ liệu *output* của tập huấn luyện cũng được sử dụng để tạo ra bộ trích chọn đặc trưng. Trong bài toán classification, việc giữ lại nhiều thông tin không quan trọng bằng việc giữ lại các

thông tin có ích cho bài toán. Ví dụ, giả sử dữ liệu thô là các hình vuông và hình tam giác có màu đỏ và xanh. Trong bài toán phân loại đa giác, nếu các nhãn là *tam giác* và *vuông*, ta không quan tâm tới màu sắc mà chỉ quan tâm tới số cạnh của đa giác. Ngược lại, trong bài toán phân loại màu, các nhãn là *xanh* và *đỏ*, ta không quan tâm tới số cạnh mà chỉ quan tâm đến màu sắc.

- **Prior knowledge about data:** Các thông tin khác đã biết về loại dữ liệu (ngoài những thông tin về raw input và output).

Sau khi các tham số mô hình của bộ feature extraction được thiết kế, dữ liệu thô ban đầu được đưa qua và tạo ra các vector đặc trưng tương ứng được gọi là *extracted feature*. Những extracted feature này sẽ được đưa vào huấn luyện các thuật toán chính như classification, regression, clustering, v.v. trong khối màu lục phía sau.

Trong một số thuật toán cao cấp hơn, việc xây dựng bộ trích chọn đặc trưng và các thuật toán chính (classification, clustering, v.v.) có thể được thực hiện cùng lúc với nhau thay vì từng bước như trên. Các mô hình đó có tên gọi chung là end-to-end. 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 vào việc hai khối phía trên được huấn luyện cùng nhau, hỗ trợ lẫn nhau cùng hướng tới mục đích cuối cùng. Thực tế cho thấy, các phương pháp state-of-the-art (các phương pháp hiệu quả nhất) thường là các mô hình end-to-end.

6.2.2 Testing phase

Khi có dữ liệu thô mới, ta sử dụng bộ trích chọn đặc trưng đã tìm được ở trên để tạo ra vector đặc trưng ứng với dữ liệu thô đó. Vector đặc trưng này được đưa vào thuật toán chính đã tìm được để đưa ra quyết định.

6.3 Một số ví dụ về Feature Engineering

6.3.1 Trực tiếp lấy dữ liệu thô

Xét một bài toán phân loại các bức ảnh xám mà mỗi bức ảnh đã có kích thước cố định là $m \times n$ pixel. Cách đơn giản nhất để tạo ra vector đặc trưng cho bức ảnh này là *kéo dài* ma trận các pixel thành một vector có mn phần tử, hay đặc trưng. Khi đó, giá trị mỗi đặc trưng sẽ là một giá trị của một pixel trong bức ảnh ban đầu, thứ tự không quan trọng. Kỹ thuật này còn được gọi là *vector hoá* (*vectorization*).

Việc làm đơn giản này đã làm mất *thông tin về không gian* (spatial information) giữa các điểm ảnh vì các pixel gần nhau theo phương ngang trong bức ảnh ban đầu có thể không còn gần nhau trong vector đặc trưng mới nữa. Tuy nhiên, trong nhiều trường hợp, kỹ thuật này vẫn mang lại kết quả khả quan.

6.3.2 Lựa chọn đặc trưng

Giả sử rằng các điểm dữ liệu có số đặc trưng khác nhau (do kích thước dữ liệu khác nhau hay do một số đặc trưng mà điểm dữ liệu này có nhưng điểm dữ liệu kia lại không thu thập được), và số lượng đặc trưng là cực lớn. Chúng ta cần *chọn* ra một số lượng nhỏ hơn các đặc trưng phù hợp với bài toán.

6.3.3 Giảm chiều dữ liệu

Một phương pháp khác thường được dùng là *làm giảm số chiều dữ liệu* (*dimensionality reduction*) để giảm bộ nhớ và khối lượng tính toán. Việc giảm số chiều này có thể được thực hiện bằng nhiều cách, trong đó *chiếu ngẫu nhiên* (*random projection*) là cách đơn giản nhất. Trong phương pháp này, một *ma trận chiếu* (*projection matrix*) ngẫu nhiên được chọn, thường là một ma trận bé—số cột nhiều hơn số hàng, để nhân vào bên trái của từng vector đặc trưng ban đầu để được các vector đặc trưng có số chiều thấp hơn. Cụ thể, giả sử vector đặc trưng ban đầu là $\mathbf{x}_0 \in \mathbb{R}^D$, nếu ta chọn một ma trận chiếu $\mathbf{P} \in \mathbb{R}^{d \times D}$ với $d \ll D$, vector mới $\mathbf{x}_1 = \mathbf{P}\mathbf{x}_0 \in \mathbb{R}^d$ có số chiều nhỏ hơn số chiều của vector \mathbf{x}_0 ban đầu.

Việc chọn một ma trận chiếu ngẫu nhiên đôi khi mang lại kết quả tệ không mong muốn vì thông tin có thể bị mất đi quá nhiều. Một phương pháp được sử dụng nhiều để tối thiểu lượng thông tin mất đi có tên là principal component analysis sẽ được trình bày trong Chương 21.

Lưu ý: Feature engineering không nhất thiết phải làm giảm số chiều dữ liệu, đôi khi vector đặc trưng có thể có kích thước lớn hơn dữ liệu thô ban đầu.

6.3.4 Bag of words

Chúng ta hẳn đã tự đặt ra các câu hỏi: với một văn bản, vector đặc trưng sẽ có dạng như thế nào? Làm sao đưa các từ, các câu, đoạn văn ở dạng *text* trong các văn bản về một vector mà mỗi phần tử là một số?

Có một kỹ thuật rất phổ biến trong xử lý văn bản có tên là *túi đựng từ* (*bag of words—BoW*).

Bắt đầu bằng ví dụ phân loại tin nhắn rác. Ta thấy rằng nếu một tin có chứa các từ *khuyến mại*, *giảm giá*, *trúng thưởng*, *miễn phí*, *quà tặng*, *tri ân*, *v.v.*, nhiều khả năng đó là một tin nhắn rác. Từ đó, phương pháp đầu tiên có thể nghĩ tới là *đếm* xem trong tin đó có bao nhiêu từ thuộc vào các từ trên, nếu số lượng này nhiều hơn một ngưỡng nào đó thì ta quyết định đó là tin rác. (Tất nhiên bài toán thực tế phức tạp hơn nhiều khi các từ có thể được viết dưới dạng không dấu, hoặc bị cố tình viết sai chính tả, hoặc dùng ngôn ngữ teen). Với các loại văn bản khác nhau, lượng từ liên quan tới từng chủ đề cũng khác nhau. Từ đó có thể dựa vào số lượng các từ trong từng loại để tạo các vector đặc trưng cho từng văn bản.

Xin lấy một ví dụ về hai văn bản đơn giản sau đây⁴:

(1) "John likes to watch movies. Mary likes movies too."

và

(2) "John also likes to watch football games."

Dựa trên hai văn bản này, ta có danh sách các từ được sử dụng, được gọi là *từ điển* (*dictionary* hoặc *codebook*) với mười từ như sau:

["John", "likes", "to", "watch", "movies", "also", "football", "games", "Mary", "too"]

Với mỗi văn bản, ta sẽ tạo ra một vector đặc trưng có số chiều bằng 10, mỗi phần tử đại diện cho số từ tương ứng xuất hiện trong văn bản đó. Với hai văn bản trên, ta sẽ có hai vector đặc trưng

(1) [1, 2, 1, 1, 2, 0, 0, 0, 1, 1]
(2) [1, 1, 1, 1, 0, 1, 1, 1, 0, 0]

Văn bản (1) có 1 từ "John", 2 từ "likes", 0 từ "also", 0 từ "football", v.v. nên ta thu được vector tương ứng như trên.

Có một vài điều cần lưu ý trong BoW:

- Với những ứng dụng thực tế, *từ điển* có nhiều hơn mười từ rất nhiều, có thể đến cả triệu, như vậy vector đặc trưng thu được sẽ rất dài. Một văn bản chỉ có một câu, và một tiểu thuyết nghìn trang đều được biểu diễn bằng các vector có kích thước như nhau.
- Có rất nhiều từ trong từ điển không xuất hiện trong một văn bản. Như vậy các vector đặc trưng thu được thường có rất nhiều phần tử bằng không. Các vector có nhiều phần tử bằng không được gọi là *vector thưa* (*sparse vector*). Để việc lưu trữ được hiệu quả hơn, ta không lưu cả vector đó mà chỉ lưu *vị trí* của các phần tử khác 0 và *giá trị* tương ứng. Chú ý rằng nếu có hơn một nửa số phần tử khác không, việc làm này lại phản tác dụng. Tuy nhiên, trường hợp này ít xảy ra vì hiếm có văn bản nào lại chứa tới một nửa từ điển.
- Ta xử lý các từ hiếm gặp không nằm trong từ điển như thế nào? Một kỹ thuật thường được dùng là thêm phần tử **<Unknown>** vào trong từ điển. Mọi từ không có trong từ điển đều được coi là **<Unknown>**. Lúc này, kích thước của vector đặc trưng sẽ tăng lên một.
- Tuy nhiên, những từ hiếm đôi khi lại mang những thông tin quan trọng nhất mà chỉ loại văn bản đó có. Đây là một nhược điểm của BoW. Có một phương pháp cải tiến giúp

⁴ *Bag of words*–Wikipedia (<https://goo.gl/rBtZqx>)

khắc phục nhược điểm này có tên là *term frequency-inverse document frequency* (TF-IDF) [SWY75] dùng để xác định tầm quan trọng của một từ trong một văn bản dựa trên toàn bộ văn bản trong cơ sở dữ liệu⁵.

- Nhược điểm lớn nhất của BoW là nó không mang thông tin về thứ tự của các từ, cũng như sự liên kết giữa các câu, các đoạn văn trong văn bản. Thứ tự của các từ trong văn bản thường mang thông tin quan trọng. Ví dụ, ba câu sau đây: “Em yêu anh không?”, “Em không yêu anh”, và “Không, (nhưng) anh yêu em” khi được trích chọn đặc trưng bằng BoW sẽ cho ra ba vector giống hệt nhau, mặc dù ý nghĩa khác hẳn nhau.

6.3.5 Bag of words trong computer vision

Bag of words cũng được áp dụng trong computer vision cho các bức ảnh với cách định nghĩa từ và từ điển khác. Xét các ví dụ sau:

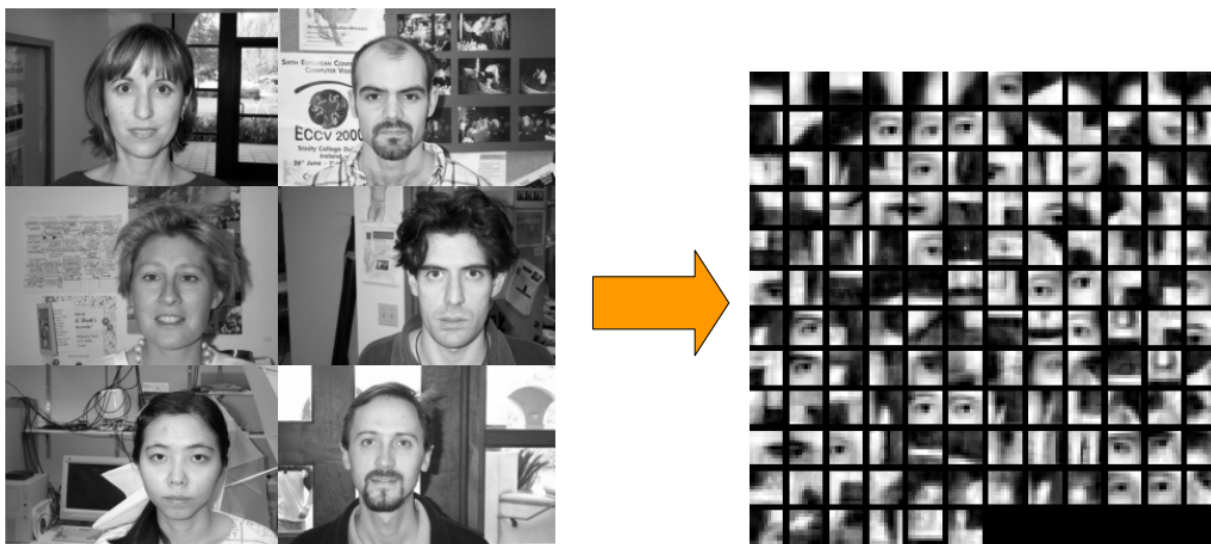
Ví dụ 1: Có hai class ảnh, một class là ảnh các khu rừng, một class là ảnh các sa mạc. Giả sử ta biết rằng một bức ảnh chỉ thuộc một trong hai loại này, việc phân loại một bức ảnh là rừng hay sa mạc một cách trực quan nhất là dựa vào màu sắc. Màu xanh lục nhiều thì là rừng, màu đỏ và vàng nhiều thì là sa mạc. Vậy chúng ta có thể có một mô hình đơn giản để trích chọn đặc trưng như sau:

- Với một bức ảnh, chuẩn bị một vector \mathbf{x} có số chiều bằng 3, đại diện cho ba màu xanh lục (x_1), đỏ (x_2), và vàng (x_3).
- Với mỗi điểm ảnh trong bức ảnh đó, xem nó gần với màu xanh, đỏ hay vàng nhất dựa trên giá trị của pixel đó. Nếu nó gần điểm xanh nhất, tăng x_1 lên một; gần đỏ nhất, tăng x_2 lên một; gần vàng nhất, tăng x_3 lên một.
- Sau khi xem xét tất cả các điểm ảnh, dù cho bức ảnh có kích thước thế nào, ta vẫn thu được một vector có kích thước bằng ba, mỗi phần tử thể hiện việc có bao nhiêu pixel trong bức ảnh có màu tương ứng. Vector cuối này còn được gọi là *histogram vector* của bức ảnh tương ứng với ba màu xanh, đỏ, vàng. Vector này có thể coi là một đặc trưng tốt trong bài toán phân lớp ảnh rừng hay say mạc.

Ví dụ 2: Trên thực tế, các bài toán xử lý ảnh không đơn giản như Ví dụ 1 trên đây. Mắt người thực ra nhạy với các đường nét, hình dáng hơn là màu sắc. Chúng ta có thể nhận biết được một bức ảnh có cây hay không ngay cả khi bức ảnh đó không có màu. Vì vậy, xem xét giá trị từng điểm ảnh một không mang lại kết quả khả quan vì lượng thông tin về đường nét bị mất quá nhiều.

Có một giải pháp là thay vì xem xét một điểm ảnh, ta xem xét một vùng hình chữ nhật nhỏ trong ảnh, vùng này còn được gọi là *patch*. Các patch này nên đủ lớn để có thể chứa được các bộ phận có thể mô tả được vật thể trong ảnh. Ví dụ với mặt người, các patch nên đủ lớn để chứa được các phần của khuôn mặt như mắt, mũi, miệng như trong Hình 6.2. Tương

⁵ 5 Algorithms Every Web Developer Can Use and Understand, section 5 (<https://goo.gl/LJW3H1>).



Hình 6.2: Bag of words cho ảnh chứa mặt người (Nguồn: *Bag of visual words model: recognizing object categories* (<https://goo.gl/EN2oSM>)).

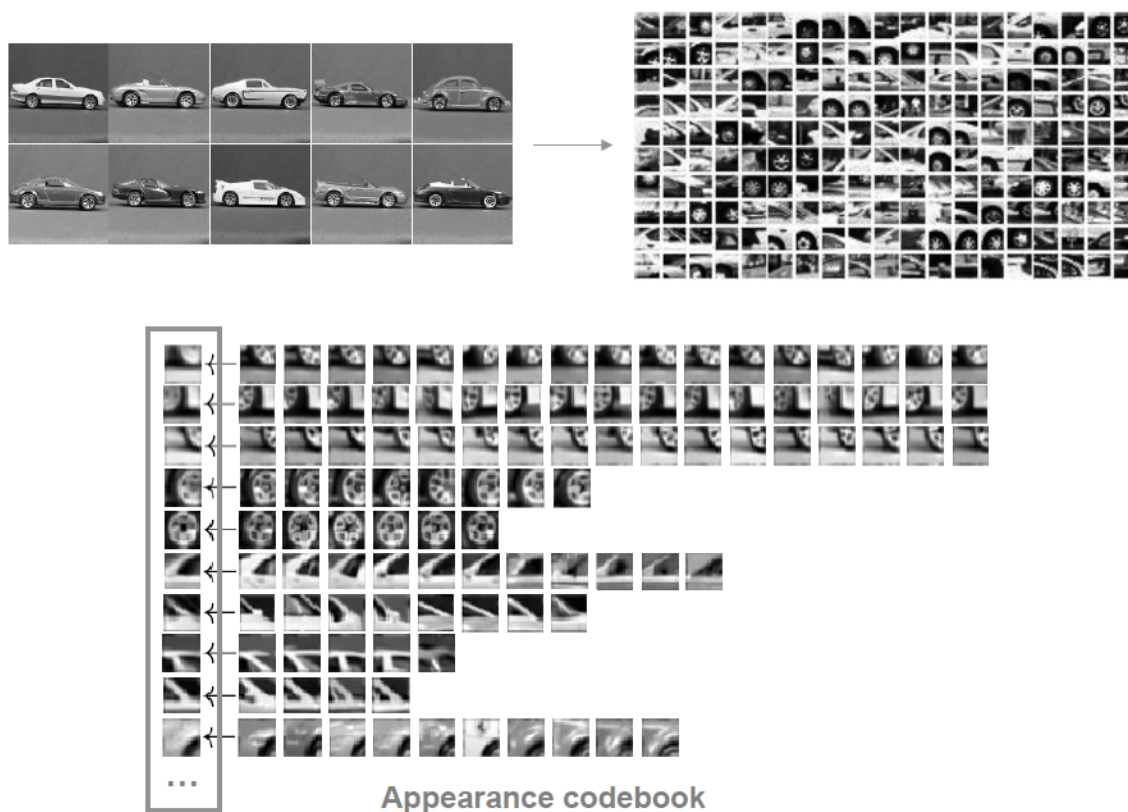
tự thể, với ảnh là ô tô, các patch thu được có thể là bánh xe, khung xe, cửa xe, v.v. trên Hình 6.3, hàng trên bên phải.

Một câu hỏi được đặt ra là, trong xử lý văn bản, hai từ được coi là như nhau nếu nó được biểu diễn bởi các ký tự giống nhau. Vậy trong xử lý ảnh, hai patch được coi là như nhau khi nào? Khi mọi pixel trong hai patch có giá trị bằng nhau sao?

Câu trả lời là không. Xác suất để hai patch giống hệt nhau từng pixel là rất thấp vì có thể một phần của vật thể trong một patch bị lệch đi vài pixel so với phần đó trong patch kia; hoặc phần vật thể trong patch bị méo, hoặc có độ sáng khác nhau, mặc dù mắt người vẫn nhìn thấy hai patch đó *rất giống nhau*. Vậy thì hai patch được coi là như nhau khi nào? Và từ điển ở đây được định nghĩa như thế nào?

Câu trả lời ngắn cho câu hỏi này là hai patch được coi là gần giống nhau nếu khoảng cách Euclid giữa hai vector tạo bởi hai patch đó là nhỏ. Từ điển sẽ có số từ do ta tự chọn. Số từ trong từ điển càng cao thì độ sai lệch càng ít, nhưng sẽ nặng về tính toán hơn.

Cụ thể hơn chúng ta có thể áp dụng một phương pháp phân nhóm đơn giản là K -means clustering (xem Chương 10). Với rất nhiều patch thu được, giả sử ta muốn xây dựng một từ điển với chỉ khoảng 1000 từ. Ta có thể dùng K -means clustering để phân toàn bộ các patch thành 1000 nhóm (bag) khác nhau. Mỗi nhóm gồm các patch gần giống nhau và được mô tả bằng trung bình cộng của tất cả các patch trong nhóm đó (xem Hình 6.3 hàng dưới). Với một ảnh bất kỳ, ta trích ra các patch từ ảnh đó, tìm xem mỗi patch gần với nhóm nào nhất trong 1000 nhóm tìm được ở trên và quyết định patch này thuộc nhóm đó. Cuối cùng, ta sẽ thu được một vector đặc trưng có kích thước bằng 1000 mà mỗi phần tử là số lượng các patch trong ảnh rơi vào nhóm tương ứng.



Source: B. Leibe

Hình 6.3: Bag of Words cho ảnh xe hơi (Nguồn: B. Leibe).

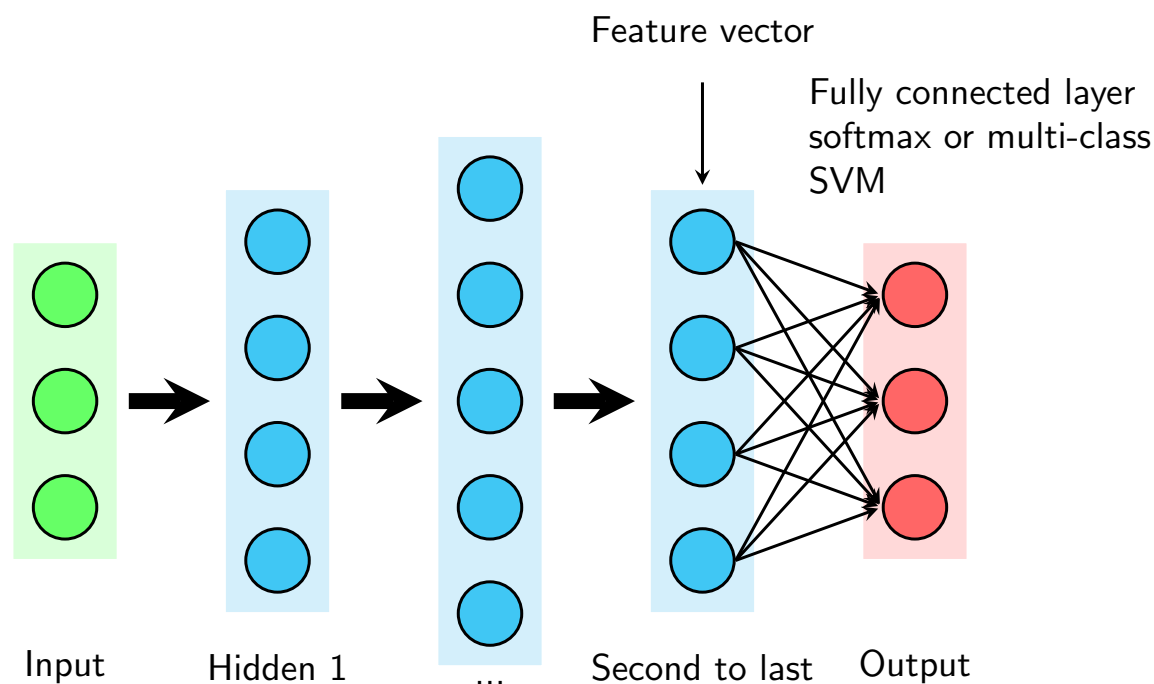
6.4 Transfer Learning cho bài toán phân loại ảnh

(Giả sử rằng bạn đọc đã có kiến thức nhất định về *deep neural network*.)

Ngoài BoW, các phương pháp thường được sử dụng để xây dựng feature vector cho ảnh là *scale invariant feature transform-SIFT* [Low99], *speeded-up robust features-SURF* [BTVG06], *histogram of oriented gradients-HOG* [DT05], *local binary pattern-LBP* [Low99], v.v.. Các bộ phân lớp thường được sử dụng là *multi-class SVM* (Chương 29), *softmax regression* (Chương 15), *sparse coding* và *discriminative dictionary learning* [WYG⁺09, VMM⁺16, VM17], *random forest* [LW⁺02], v.v..

Các feature được tạo bởi các phương pháp nêu trên thường được gọi là các *feature được tạo thủ công (hand-crafted feature)* vì chúng chủ yếu dựa trên các quan sát về đặc tính riêng của ảnh. Các phương pháp này cho kết quả khá ấn tượng trong một số trường hợp. Tuy nhiên, chúng vẫn còn nhiều hạn chế vì quá trình tìm ra các feature và các classifier phù hợp vẫn là riêng biệt.

Những năm gần đây, *deep learning* phát triển cực nhanh dựa trên lượng dữ liệu huấn luyện khổng lồ và khả năng tính toán ngày càng được cải tiến của các máy tính. Các kết quả cho



Hình 6.4: Mô hình chung cho các bài toán classification sử dụng Deep Learning. Layer cuối cùng thường là một Fully Connected Layer và thường là một Softmax Regression.

bài toán phân loại ảnh ngày càng được nâng cao. Bộ cơ sở dữ liệu thường được dùng nhất là ImageNet (<https://www.image-net.org>) 1.2 triệu ảnh cho 1000 class khác nhau. Rất nhiều các mô hình deep learning đã giành chiến thắng trong các cuộc thi *ImageNet large scale visual recognition challenge-ILSVRC* : AlexNet [KSH12], ZFNet [ZF14], GoogLeNet [SLJ⁺15], ResNet [HZRS16], VGG [SZ14]. Nhìn chung, các mô hình này là các neural network với rất nhiều layer (xem Chương 16). Các layer phía trước thường là các convolutional layer. Layer cuối cùng là một fully connected layer và thường là một softmax regression (xem Hình 6.4). Số lượng unit ở layer cuối cùng bằng với số lượng class (với ImageNet là 1000). Vì vậy output ở layer gần cuối cùng (*second to last layer*) có thể được coi là feature vector và softmax regression chính là classifier được sử dụng.

Chính nhờ việc các feature và classifier được huấn luyện cùng nhau thông qua việc tối ưu các hệ số trong deep network khiến cho các mô hình này đạt kết quả tốt. Tuy nhiên, những mô hình này đều bao gồm rất nhiều layer và các trọng số. Việc huấn luyện dựa trên 1.2M bức ảnh của ImageNet cũng tốn rất nhiều thời gian (2-3 tuần).

Với các bài toán dựa trên tập dữ liệu khác với ít dữ liệu hơn, ta có thể không cần xây dựng lại network và huấn luyện nó từ đầu. Thay vào đó, ta có thể sử dụng các mô hình đã được huấn luyện nêu trên, và sử dụng một vài kỹ thuật khác để giải quyết bài toán. Phương pháp sử dụng các mô hình có sẵn như thế này được gọi là *transfer learning*.

Như đã đề cập, toàn bộ các layer trừ output layer có thể được coi là một feature extractor. Dựa trên nhận xét rằng các bức ảnh đều có những đặc tính giống nhau nào đó, với cơ sở dữ

liệu khác, ta cũng có thể sử dụng phần feature extractor này để tạo ra các feature vector. Sau đó, ta thay output layer cũng bằng một softmax regression, multi-class SVM (với số lượng unit phù hợp) hoặc các classifier phổ biến khác. Cách làm này có thể tăng độ chính xác phân lớp lên đáng kể so với việc sử dụng các *hand-crafted features*.

Hướng tiếp cận thứ hai là sử dụng các mô hình đã được huấn luyện và cho cập nhật một vài layer cuối dựa trên dữ liệu mới thêm một vài vòng lặp. Kỹ thuật này được gọi là *fine-tuning*. Việc này được dựa trên quan sát rằng những layer đầu trong deep network thường giúp trích xuất những đặc tính chung của ảnh (các cạnh, còn được gọi là *low-level feature*), các layer cuối thường mang những đặc trưng riêng của cơ sở dữ liệu (CSDL) (và được gọi là *high-level feature*). Vì vậy, việc huấn luyện các layer cuối mang nhiều giá trị hơn.

Dựa trên kích thước và độ tương quan giữa CSDL mới và CSDL gốc (chủ yếu là ImageNet), có một vài quy tắc để huấn luyện network mới như sau⁶:

- *CSDL mới là nhỏ và tương tự như CSDL gốc.* Vì CSDL mới nhỏ, việc tiếp tục huấn luyện mô hình có thể dễ dẫn đến hiện tượng overfitting (Chương 8). Cũng vì hai CSDL là tương tự nhau, ta dự đoán rằng các high-level feature của chúng là tương tự nhau. Vì vậy, ta không cần huấn luyện lại network mà chỉ cần huấn luyện một classifier dựa trên feature vector ở đầu ra ở layer gần cuối.
- *CSDL mới là lớn và tương tự như CSDL gốc.* Vì CSDL này lớn, overfitting ít có khả năng xảy ra hơn, ta có thể huấn luyện mô hình thêm một vài vòng lặp. Việc huấn luyện có thể được thực hiện trên toàn bộ hoặc chỉ một vài layer cuối.
- *CSDL mới là nhỏ và rất khác với CSDL gốc.* Vì CSDL này nhỏ, tốt hơn hết là dùng các classifier đơn giản để tránh overfitting. Nếu muốn huấn luyện thêm, ta cũng chỉ nên thực hiện trên các layer cuối. Hoặc sử dụng một kỹ thuật khác là coi đầu ra của một layer *xa* layer cuối hơn (xa hơn layer gần cuối) làm các feature vector.
- *CSDL mới là lớn và rất khác CSDL gốc.* Thực tế cho thấy, sử dụng các network sẵn có trên CSDL mới vẫn hữu ích. Trong trường hợp này, ta vẫn có thể sử dụng các network sẵn có như là điểm khởi tạo của network mới, không nên huấn luyện network mới từ đầu.

Có một điểm đáng chú ý nữa là khi tiếp tục huấn luyện các network này, ta chỉ nên chọn *learning rate* nhỏ để các hệ số mới không đi quá xa so với các hệ số đã được huấn luyện ở các mô hình trước.

6.5 Chuẩn hoá vector đặc trưng

Các điểm dữ liệu đôi khi được đo đạc với những đơn vị khác nhau, mét và feet chẳng hạn. Hoặc có hai thành phần (của vector dữ liệu) chênh lệch nhau quá lớn, một thành phần có khoảng giá trị từ 0 đến 1000, thành phần kia chỉ có khoảng giá trị từ 0 đến 1 chẳng hạn. Lúc này, chúng ta cần chuẩn hóa dữ liệu trước khi thực hiện các bước tiếp theo.

⁶ *Transfer Learning, CS231n* (<https://goo.gl/VN1g7F>)

Chú ý: việc chuẩn hóa này chỉ được thực hiện khi vector dữ liệu đã có cùng chiều.

Một vài phương pháp chuẩn hóa thường dùng:

6.5.1 Rescaling

Phương pháp đơn giản nhất là đưa tất cả các đặc trưng về cùng một khoảng, chẳng hạn $[0, 1]$ hoặc $[-1, 1]$ tùy thuộc vào ứng dụng. Nếu muốn đưa đặc trưng thứ i của một vector đặc trưng \mathbf{x} về khoảng $[0, 1]$, công thức sẽ là:

$$x'_i = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$$

trong đó x_i và x'_i lần lượt là giá trị đặc trưng ban đầu và giá trị đặc trưng sau khi được chuẩn hóa. $\min(x_i), \max(x_i)$ là giá trị nhỏ nhất và lớn nhất của đặc trưng thứ i xét trên toàn bộ các điểm dữ liệu của tập huấn luyện.

6.5.2 Standardization

Một phương pháp khác cũng thường được sử dụng là giả sử mỗi đặc trưng đều có phân phối chuẩn với kỳ vọng là 0 và phương sai là 1. Khi đó, công thức chuẩn hóa sẽ là

$$x'_i = \frac{x_i - \bar{x}_i}{\sigma_i}$$

với \bar{x}_i, σ_i lần lượt là kỳ vọng và độ lệch chuẩn (*standard deviation*) của đặc trưng đó xét trên toàn bộ dữ liệu huấn luyện.

6.5.3 Scaling to unit length

Một lựa chọn khác cũng được sử dụng rộng rãi là chuẩn hóa các thành phần của mỗi vector dữ liệu sao cho toàn bộ vector có độ dài Euclid bằng một. Việc này có thể được thực hiện bằng cách chia mỗi vector đặc trưng cho ℓ_2 norm của nó:

$$\mathbf{x}' = \frac{\mathbf{x}}{\|\mathbf{x}\|_2}$$

6.6 Đọc thêm

1. G. Csurka *et al.*, *Visual categorization with bags of keypoints*. Workshop on statistical learning in computer vision, ECCV. Vol. 1. No. 1-22. 2004 [CDF+04].
2. S. Lazebnik *et al.*, *Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories.*, CVPR 2006 [LSP06]
3. *Preprocessing data, scikit learn* (<https://goo.gl/gkCuUp>).

Linear regression

Trong chương này, chúng ta cùng làm quen với một trong những thuật toán machine learning cơ bản nhất—linear regression (hồi quy tuyến tính). Qua chương này, bạn đọc sẽ có cái nhìn ban đầu về việc xây dựng một hệ thống machine learning. Linear regression là một thuật toán supervised, ở đó quan hệ giữa đầu vào và đầu ra được mô tả bởi một hàm tuyến tính. Thuật toán này còn được gọi là *linear fitting* hoặc *linear least square*.

7.1 Giới thiệu

Xét bài toán ước lượng giá của một căn nhà rộng x_1 m², có x_2 phòng ngủ và cách trung tâm thành phố x_3 km. Giả sử ta đã thu thập được số liệu từ 1000 căn nhà trong thành phố đó, liệu rằng khi có một căn nhà mới với các thông số về diện tích x_1 , số phòng ngủ x_2 và khoảng cách tới trung tâm x_3 , chúng ta có thể dự đoán được giá y của căn nhà đó không? Nếu có thì hàm dự đoán $y = f(\mathbf{x})$ sẽ có dạng như thế nào. Ở đây, vector đặc trưng $\mathbf{x} = [x_1, x_2, x_3]^T$ là một vector cột chứa thông tin đầu vào, đầu ra y là một số vô hướng.

Một cách trực quan, ta có thể thấy rằng: (i) diện tích nhà càng lớn thì giá nhà càng cao; (ii) số lượng phòng ngủ càng lớn thì giá nhà càng cao; (iii) càng xa trung tâm thì giá nhà càng giảm. Dựa trên quan sát này, ta có thể mô hình quan hệ giữa đầu ra và đầu vào bằng một hàm tuyến tính đơn giản:

$$y \approx \hat{y} = f(\mathbf{x}) = w_1x_1 + w_2x_2 + w_3x_3 = \mathbf{x}^T \mathbf{w} \quad (7.1)$$

trong đó $\mathbf{w} = [w_1, w_2, w_3]^T$ là vector hệ số (hoặc trọng số—*weight vector*) ta cần đi tìm. Đây cũng chính là tham số mô hình của bài toán. Mối quan hệ $y \approx f(\mathbf{x})$ như trong (7.1) là một mối quan hệ tuyến tính.

Bài toán trên đây là bài toán dự đoán giá trị của đầu ra dựa trên vector đặc trưng đầu vào. Ngoài ra, giá trị của đầu ra có thể nhận rất nhiều giá trị thực dương khác nhau. Vì vậy, đây là một bài toán regression. Mối quan hệ $\hat{y} = \mathbf{x}^T \mathbf{w}$ là một mối quan hệ tuyến tính. Tên gọi *linear regression* xuất phát từ đây.

Chú ý:

1. y là giá trị thực của đầu ra (*ground truth*), trong khi \hat{y} là giá trị đầu ra dự đoán (*predicted output*) của mô hình linear regression. Nhìn chung, y và \hat{y} là hai giá trị khác nhau do có sai số mô hình, tuy nhiên, chúng ta mong muốn rằng sự khác nhau này rất nhỏ.
2. *Linear* hay *tuyến tính* hiểu một cách đơn giản là *thẳng, phẳng*. Trong không gian hai chiều, một hàm số được gọi là *tuyến tính* nếu đồ thị của nó có dạng một *đường thẳng*. Trong không gian ba chiều, một hàm số được gọi là *tuyến tính* nếu đồ thị của nó có dạng một *mặt phẳng*. Trong không gian nhiều hơn ba chiều, khái niệm *mặt phẳng* không còn phù hợp nữa, thay vào đó, một khái niệm khác ra đời được gọi là *siêu mặt phẳng* (*hyperplane*). Các hàm số tuyến tính là các hàm đơn giản nhất, vì chúng thuận tiện trong việc hình dung và tính toán.

7.2 Xây dựng và tối ưu hàm mất mát

7.2.1 Sai số dự đoán

Sau khi đã xây dựng được mô hình dự đoán đầu ra như (7.1), ta cần tìm một phép đánh giá phù hợp với bài toán. Với bài toán regression nói chung, ta mong muốn rằng sự sai khác e giữa giá trị thực y và giá trị dự đoán \hat{y} là nhỏ nhất. Nói cách khác, chúng ta muốn giá trị sau đây càng nhỏ càng tốt:

$$\frac{1}{2}e^2 = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}(y - \mathbf{x}^T \mathbf{w})^2 \quad (7.2)$$

ở đây ta lấy bình phương vì $e = y - \hat{y}$ có thể là một số âm. Việc sai số là nhỏ nhất có thể được mô tả bằng cách lấy trị tuyệt đối $|e| = |y - \hat{y}|$, tuy nhiên, cách làm này ít được sử dụng vì hàm trị tuyệt đối không khả vi tại mọi điểm, không thuận tiện cho việc tối ưu sau này. Hệ số $\frac{1}{2}$ sẽ bị triệt tiêu sau này khi lấy đạo hàm của e theo tham số mô hình \mathbf{w} .

7.2.2 Hàm mất mát

Điều tương tự xảy ra với tất cả các cặp (*input, output*) $(\mathbf{x}_i, y_i), i = 1, 2, \dots, N$, với N là số lượng dữ liệu quan sát được. Điều chúng ta mong muốn—trung bình sai số là nhỏ nhất—tương đương với việc tìm \mathbf{w} để hàm số sau đạt giá trị nhỏ nhất:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^N (y_i - \mathbf{x}_i^T \mathbf{w})^2 \quad (7.3)$$

Hàm số $\mathcal{L}(\mathbf{w})$ chính là hàm mất mát của linear regression với tham số mô hình $\theta = \mathbf{w}$. Chúng ta luôn mong muốn rằng sự mất mát là nhỏ nhất, điều này có thể đạt được bằng cách tối thiểu hàm mất mát theo \mathbf{w} :

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathbf{w}) \quad (7.4)$$

\mathbf{w}^* là nghiệm cần tìm, đôi khi dấu $*$ được bỏ đi và nghiệm có thể được viết gọn lại thành $\mathbf{w} = \underset{\mathbf{w}}{\operatorname{argmin}} \mathcal{L}(\mathbf{w})$.

Trung bình sai số

Việc lấy trung bình (hệ số $\frac{1}{N}$) hay lấy tổng trong hàm mất mát, về mặt toán học, không ảnh hưởng tới nghiệm của bài toán. Trong machine learning, các hàm mất mát thường có chứa hệ số tính trung bình theo từng điểm dữ liệu trong tập huấn luyện. Khi tính giá trị của hàm mất mát trên tập kiểm thử, ta cũng tính trung bình lỗi của mỗi điểm. Việc lấy trung bình này quan trọng vì số lượng điểm dữ liệu trong mỗi tập dữ liệu có thể thay đổi. Việc tính toán mất mát trên từng điểm dữ liệu sẽ hữu ích hơn trong việc đánh giá chất lượng mô hình sau này. Ngoài ra, việc lấy trung bình cũng giúp tránh hiện tượng tràn số khi số lượng điểm dữ liệu quá nhiều.

Trước khi đi xây dựng nghiệm cho bài toán tối ưu hàm mất mát, ta thấy rằng hàm số này có thể được viết gọn lại dưới dạng ma trận, vector, và norm như dưới đây:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \sum_{i=1}^N (y_i - \mathbf{x}_i^T \mathbf{w})^2 = \frac{1}{2N} \left\| \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} - \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} \mathbf{w} \right\|_2^2 = \frac{1}{2N} \|\mathbf{y} - \mathbf{X}^T \mathbf{w}\|_2^2 \quad (7.5)$$

với $\mathbf{y} = [y_1, y_2, \dots, y_N]^T$, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]$. Như vậy $\mathcal{L}(\mathbf{w})$ là một hàm số liên quan tới bình phương của ℓ_2 norm.

7.2.3 Nghiệm cho bài toán Linear Regression

Nhận thấy rằng hàm mất mát $\mathcal{L}(\mathbf{w})$ có đạo hàm tại mọi \mathbf{w} (xem Bảng 2.1). Vậy việc tìm giá trị tối ưu của \mathbf{w} có thể được thực hiện thông qua việc giải phương trình đạo hàm của $\mathcal{L}(\mathbf{w})$ theo \mathbf{w} bằng không. Thật may mắn, đạo hàm của hàm mất mát của linear regression rất đơn giản:

$$\frac{\nabla \mathcal{L}(\mathbf{w})}{\nabla \mathbf{w}} = \frac{1}{N} \mathbf{X}(\mathbf{X}^T \mathbf{w} - \mathbf{y}) \quad (7.6)$$

Giải phương trình đạo hàm bằng không:

$$\frac{\nabla \mathcal{L}(\mathbf{w})}{\nabla \mathbf{w}} = \mathbf{0} \Leftrightarrow \mathbf{X}\mathbf{X}^T \mathbf{w} = \mathbf{X}\mathbf{y} \quad (7.7)$$

Nếu ma trận $\mathbf{X}\mathbf{X}^T$ khả nghịch, phương trình (7.7) có nghiệm duy nhất $\mathbf{w} = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X}\mathbf{y}$.

Nếu ma trận $\mathbf{X}\mathbf{X}^T$ không khả nghịch, phương trình (7.7) sẽ vô nghiệm hoặc có vô số nghiệm. Lúc này, một nghiệm đặc biệt của phương trình có thể được xác định dựa vào *giả nghịch đảo* (pseudo inverse). Người ta chứng minh được rằng¹ với mọi ma trận \mathbf{X} , luôn tồn tại duy nhất

¹ Least Squares, Pseudo-Inverse, PCA & SVD (<https://goo.gl/RoQ6mS>)

một giá trị \mathbf{w} có ℓ_2 norm nhỏ nhất giúp tối thiểu $\|\mathbf{X}^T \mathbf{w} - \mathbf{y}\|_F^2$. Cụ thể, $\mathbf{w} = (\mathbf{X}\mathbf{X}^T)^\dagger \mathbf{X}\mathbf{y}$, trong đó $(\mathbf{X}\mathbf{X}^T)^\dagger$ là giả nghịch đảo của $\mathbf{X}\mathbf{X}^T$. Giả nghịch đảo của một ma trận \mathbf{A} luôn luôn tồn tại, thậm chí cả khi ma trận đó không vuông. Khi \mathbf{A} là vuông và khả nghịch thì giả nghịch đảo chính là nghịch đảo. Nghiệm của bài toán tối ưu (7.4) là

$$\mathbf{w} = (\mathbf{X}\mathbf{X}^T)^\dagger \mathbf{X}\mathbf{y} \quad (7.8)$$

Hàm số tính giả nghịch đảo của một ma trận bất kỳ có sẵn trong thư viện numpy.

7.2.4 Bias trick

Chú ý rằng quan hệ $y = \mathbf{x}^T \mathbf{w}$ là một quan hệ tuyến tính. Linear regression thường đề nói tới một quan hệ hơi phức tạp hơn một chút khi có sự xuất hiện của một số hạng tự do b :

$$y = \mathbf{x}^T \mathbf{w} + b \quad (7.9)$$

Mọi quan hệ này còn được gọi là *affine*. Nếu $b = 0$, đường thẳng $\mathbf{y} = \mathbf{x}^T \mathbf{w} + b$ luôn đi qua gốc toạ độ. Việc thêm hệ số b vào sẽ khiến cho mô hình linh hoạt hơn một chút bằng cách bỏ ràng buộc đường thẳng quan hệ giữa đầu ra và đầu vào luôn đi qua gốc toạ độ. Đại lượng b còn được gọi là *bias*. Đại lượng này cũng có thể *học được* như vector hệ số \mathbf{w} .

Để ý thấy rằng, nếu coi $x_0 = 1$, ta sẽ có:

$$y = \mathbf{x}^T \mathbf{w} + b = w_1 x_1 + w_2 x_2 + \dots + w_d x_d + b x_0 = \bar{\mathbf{x}}^T \bar{\mathbf{w}} \quad (7.10)$$

trong đó $\bar{\mathbf{x}} = [x_0, x_1, x_2, \dots, x_N]^T$ và $\bar{\mathbf{w}} = [b, w_1, w_2, \dots, w_N]$. Nếu đặt $\bar{\mathbf{X}} = [\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \dots, \bar{\mathbf{x}}_N]$ ta sẽ có nghiệm của bài toán tối thiểu hàm mất mát:

$$\bar{\mathbf{w}} = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{2N} \|\mathbf{y} - \bar{\mathbf{X}}^T \bar{\mathbf{w}}\|_2^2 = (\bar{\mathbf{X}}\bar{\mathbf{X}}^T)^\dagger \bar{\mathbf{X}}\mathbf{y} \quad (7.11)$$

Kỹ thuật thêm một đặc trưng bằng 1 vào vector đặc trưng và ghép bias b vào vector hệ số \mathbf{w} như trên còn được gọi là *bias trick*. Chúng ta sẽ còn gặp lại kỹ thuật này nhiều lần trong cuốn sách này.

7.3 Ví dụ trên Python

7.3.1 Bài toán

Xét một ví dụ đơn giản có thể áp dụng linear regression. Chúng ta cũng sẽ so sánh nghiệm của bài toán khi giải theo phương trình (7.11) và nghiệm tìm được khi dùng thư viện scikit-learn của Python.

Xét ví dụ về dự đoán cân nặng dựa theo chiều cao. Xét bảng cân nặng và chiều cao của 15 người trong Bảng 7.1. Dữ liệu của hai người có chiều cao 155 cm và 160 cm được tách ra làm test set, phần còn lại tạo thành training set.

Bài toán đặt ra là: liệu có thể dự đoán cân nặng của một người dựa vào chiều cao của họ không? (*Trên thực tế, tất nhiên là không, vì cân nặng còn phụ thuộc vào nhiều yếu tố khác nữa, thể tích chẳng hạn*). Có thể thấy là cân nặng thường tỉ lệ thuận với chiều cao (càng cao càng nặng), nên có thể sử dụng linear regression cho việc dự đoán này.

Bảng 7.1: Bảng dữ liệu về chiều cao và cân nặng của 15 người

Chiều cao (cm)	Cân nặng (kg)	Chiều cao (cm)	Cân nặng (kg)
147	49	168	60
150	50	170	72
153	51	173	63
155	52	175	64
158	54	178	66
160	56	180	67
163	58	183	68
165	59		

7.3.2 Hiện thị dữ liệu trên đồ thị

Trước tiên, chúng ta khai báo training data:

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
# height (cm), input data, each row is a data point
X = np.array([[147, 150, 153, 158, 163, 165, 168, 170, 173, 175, 178, 180, 183]]).T
# weight (kg)
y = np.array([ 49, 50, 51, 54, 58, 59, 60, 62, 63, 64, 66, 67, 68])
```

Các điểm dữ liệu được minh họa bởi các điểm màu đỏ trong Hình 7.1 Ta thấy rằng dữ liệu được sắp xếp gần như theo một đường thẳng, vậy mô hình linear regression sau đây có khả năng cho kết quả tốt:

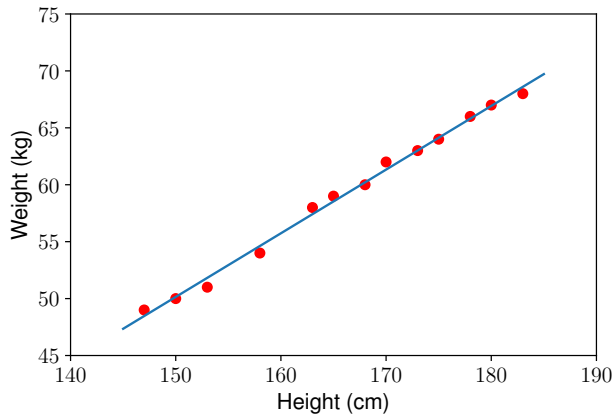
$$(\text{cân nặng}) = w_1 * (\text{chiều cao}) + w_0$$

ở đây w_0 chính là bias b .

7.3.3 Nghiệm theo công thức

Tiếp theo, chúng ta sẽ tính toán các hệ số w_1 và w_0 dựa vào công thức (7.11). Chú ý: giả nghịch đảo của một ma trận A trong Python sẽ được tính bằng `numpy.linalg.pinv(A)`.

```
# Building Xbar
one = np.ones((X.shape[0], 1))
Xbar = np.concatenate((one, X), axis = 1) # each point is one row
# Calculating weights of the fitting line
A = np.dot(Xbar.T, Xbar)
b = np.dot(Xbar.T, y)
w = np.dot(np.linalg.pinv(A), b)
# weights
w_0, w_1 = w[0], w[1]
```



Hình 7.1: Phân bố của các điểm dữ liệu (màu đỏ) và đường thẳng xấp xỉ tìm được bởi linear regression.

Đường thẳng mô tả mối quan hệ giữa đầu vào và đầu ra được cho trên Hình 7.1. Ta thấy rằng các điểm dữ liệu màu đỏ nằm khá gần đường thẳng dự đoán màu xanh. Vậy mô hình linear regression hoạt động tốt với tập dữ liệu huấn luyện. Bây giờ, chúng ta sử dụng mô hình này để dự đoán dữ liệu trong test set:

```
y1 = w_1*155 + w_0
y2 = w_1*160 + w_0
print('Input 155cm, true output 52kg, predicted output %.2fkg' % (y1) )
print('Input 160cm, true output 56kg, predicted output %.2fkg' % (y2) )
```

Kết quả:

```
Input 155cm, true output 52kg, predicted output 52.94kg
Input 160cm, true output 56kg, predicted output 55.74kg
```

Chúng ta thấy rằng kết quả dự đoán khá gần với số liệu thực tế.

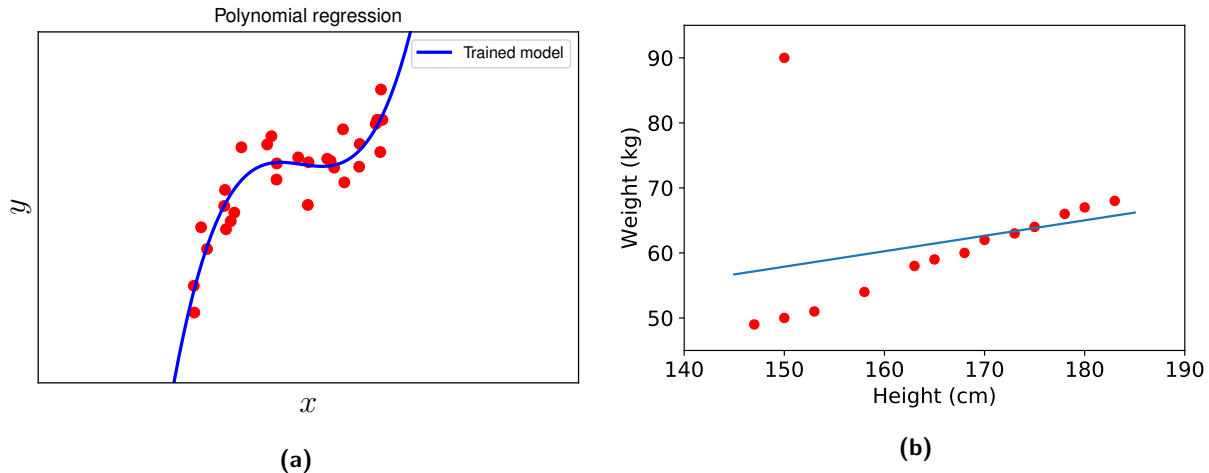
7.3.4 Nghiệm theo thư viện scikit-learn

Tiếp theo, chúng ta sẽ sử dụng thư viện scikit-learn để tìm nghiệm.

```
from sklearn import datasets, linear_model
# fit the model by Linear Regression
regr = linear_model.LinearRegression()
regr.fit(X, y) # in scikit-learn, each sample is one row
# Compare two results
print("scikit-learn's solution : w_1 = ", regr.coef_[0], "w_0 = ", regr.intercept_)
print("our solution : w_1 = ", w[1], "w_0 = ", w[0])
```

Kết quả dưới đây cho thấy rằng hai cách tính cho kết quả như nhau.

```
scikit-learn solution : w_1 = [ 0.55920496] w_0 = [-33.73541021]
our solution : w_1 = [ 0.55920496] w_0 = [-33.73541021]
```

Hình 7.2: (a) Polynomial regression bậc ba. (b) Linear regression nhạy cảm với nhiễu nhỏ.

7.4 Thảo luận

7.4.1 Các bài toán có thể giải bằng linear regression

Hàm số $y \approx f(\mathbf{x}) = \mathbf{x}^T \mathbf{w}$ là một hàm tuyến tính theo cả \mathbf{w} và \mathbf{x} . Trên thực tế, linear regression có thể áp dụng cho các mô hình chỉ cần tuyến tính theo \mathbf{w} . Ví dụ,

$$y \approx w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 \sin(x_2) + w_5 x_1 x_2 + w_0 \quad (7.12)$$

là một hàm tuyến tính theo \mathbf{w} và vì vậy cũng có thể được giải bằng linear regression. Với mỗi vector đặc trưng $\mathbf{x} = [x_1, x_2]^T$, chúng ta tính toán vector đặc trưng mới $\tilde{\mathbf{x}} = [x_1, x_2, x_1^2, \sin(x_2), x_1 x_2]^T$ rồi áp dụng linear regression với dữ liệu mới này. Tuy nhiên, việc tìm ra các hàm số $\sin(x_2)$ hay $x_1 x_2$ là tương đối *không tự nhiên*. *Hồi quy đa thức (polynomial regression)* thường được sử dụng nhiều hơn với các vector đặc trưng mới có dạng $[1, x_1, x_1^2, \dots]^T$. Một ví dụ về hồi quy đa thức bậc 3 được thể hiện trong Hình 7.2a.

7.4.2 Hạn chế của linear regression

Hạn chế đầu tiên của linear regression là nó rất **nhạy cảm với nhiễu** (*sensitive to noise*). Trong ví dụ về mối quan hệ giữa chiều cao và cân nặng bên trên, nếu có chỉ một cặp dữ liệu *nh nhiễu* (150 cm, 90kg) thì kết quả sẽ sai khác đi rất nhiều (xem Hình 7.2b).

Vì vậy, trước khi thực hiện linear regression, các nhiễu cần phải được loại bỏ. Bước này được gọi là *tiền xử lý (pre-processing)*. Hoặc hàm mất mát có thể thay đổi một chút để tránh việc tối ưu các nhiễu bằng cách sử dụng *Huber loss* (<https://goo.gl/TBUWzg>). Linear regression với Huber loss được gọi là *Huber regression*, được khẳng định là *robust to noise* (ít bị ảnh hưởng hơn bởi nhiễu). Xem thêm *Huber Regressor, scikit learn* (<https://goo.gl/h2rKu5>).

Hạn chế thứ hai của linear regression là nó **không biểu diễn được các mô hình phức tạp**. Mặc dù trong phần trên, chúng ta thấy rằng phương pháp này có thể được áp dụng nếu quan hệ giữa *outcome* và *input* không nhất thiết phải là tuyến tính, nhưng mối quan

hệ này vẫn đơn giản nhiều so với các mô hình thực tế. Hơn nữa, việc tìm ra các đặc trưng $x_1^2, \sin(x_2), x_1x_2$ như ở trên là ít khả thi.

7.4.3 Ridge regression

Trong trường hợp ma trận \mathbf{XX}^T không khả nghịch, có một kỹ thuật nhỏ để tránh hiện tượng này là biến đổi \mathbf{XX}^T một chút để biến nó trở thành $\mathbf{A} = \mathbf{XX}^T + \lambda\mathbf{I}$ với λ là một số dương rất nhỏ và \mathbf{I} là ma trận đơn vị với bậc phù hợp.

Ma trận \mathbf{A} là khả nghịch vì nó là một ma trận xác định dương. Thật vậy, với mọi $\mathbf{w} \neq \mathbf{0}$,

$$\mathbf{w}^T \mathbf{A} \mathbf{w} = \mathbf{w}^T (\mathbf{XX}^T + \lambda\mathbf{I}) \mathbf{w} = \mathbf{w}^T \mathbf{XX}^T \mathbf{w} + \lambda \mathbf{w}^T \mathbf{w} = \|\mathbf{X}^T \mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2 > 0 \quad (7.13)$$

Lúc này, nghiệm của bài toán là $\mathbf{y} = (\mathbf{XX}^T + \lambda\mathbf{I})^{-1} \mathbf{X} \mathbf{y}$. Nếu xét hàm mất mát

$$\mathcal{L}_2(\mathbf{w}) = \frac{1}{2N} (\|\mathbf{y} - \mathbf{X}^T \mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2) \quad (7.14)$$

với phương trình đạo hàm theo \mathbf{w} bằng không:

$$\frac{\nabla \mathcal{L}_2(\mathbf{w})}{\nabla \mathbf{w}} = \mathbf{0} \Leftrightarrow \frac{1}{N} (\mathbf{X}(\mathbf{X}^T \mathbf{w} - \mathbf{y}) + \lambda \mathbf{w}) = \mathbf{0} \Leftrightarrow (\mathbf{XX}^T + \lambda\mathbf{I}) \mathbf{w} = \mathbf{X} \mathbf{y} \quad (7.15)$$

Ta thấy $\mathbf{w} = (\mathbf{XX}^T + \lambda\mathbf{I})^{-1} \mathbf{X} \mathbf{y}$ chính là nghiệm của bài toán tối thiểu $\mathcal{L}_2(\mathbf{w})$ trong (7.14). Mô hình machine learning với hàm mất mát (7.14) còn được gọi là *ridge regression*. Ngoài việc giúp cho phương trình đạo hàm theo hệ số bằng không có nghiệm duy nhất, ridge regression còn giúp cho mô hình tránh được overfitting như chúng ta sẽ thấy ở Chương 8.

7.4.4 Phương pháp tối ưu khác

Linear regression là một mô hình đơn giản, lời giải cho phương trình đạo hàm bằng không cũng khá đơn giản. Trong hầu hết các trường hợp, chúng ta không thể giải được phương trình đạo hàm bằng không. Tuy nhiên, nếu một hàm mất mát có đạo hàm không quá phức tạp, nó có thể được giải bằng một phương pháp rất hữu dụng có tên là gradient descent. Trên thực tế, một vector đặc trưng có thể có kích thước rất lớn, dẫn đến ma trận \mathbf{XX}^T cũng có kích thước lớn và việc tính ma trận nghịch đảo có thể không lợi về mặt tính toán. Gradient descent sẽ giúp tránh được việc tính ma trận nghịch đảo. Chúng ta sẽ hiểu kỹ hơn về phương pháp này trong Chương 12.

7.4.5 Đọc thêm

1. *Simple Linear Regression Tutorial for Machine Learning* (<https://goo.gl/WRVda8>).
2. *Regularization: Ridge Regression and the LASSO* (<https://goo.gl/uRzN1K>).

Overfitting

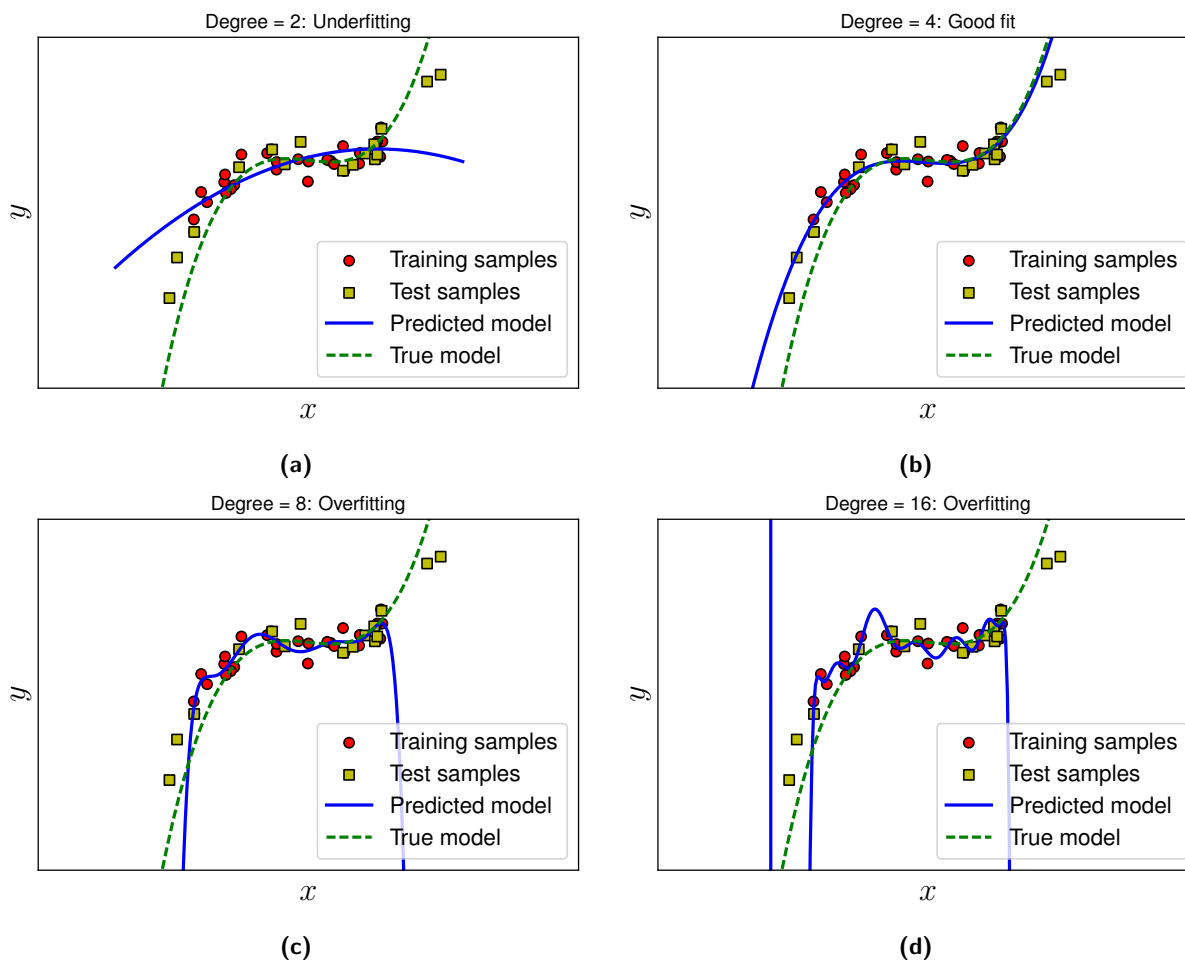
Overfitting là một hiện tượng không mong muốn thường gặp, người xây dựng mô hình machine learning cần nắm được các kỹ thuật để tránh hiện tượng này.

8.1 Giới thiệu

Trong các bài toán supervised learning, chúng ta thường phải đi tìm một mô hình ánh xạ các vector đặc trưng thành các kết quả tương ứng trong training set. Tức là đi tìm hàm số f sao cho $y_i \approx f(\mathbf{x}_i)$, $\forall i = 1, 2, \dots, N$. Một cách tự nhiên, ta sẽ đi tìm các tham số mô hình của f sao cho việc xấp xỉ có sai số càng nhỏ càng tốt. Nói cách khác, mô hình càng *khớp* (*fit*) với dữ liệu càng tốt. Tuy nhiên, sự thật là nếu một mô hình *quá fit* với dữ liệu thì nó sẽ gây phản tác dụng. Hiện tượng *quá fit* này trong machine learning được gọi là *overfitting*. Đây là một hiện tượng xấu cần tránh. Vì có thể mô hình rất *fit* với training set nhưng lại không biểu diễn tốt dữ liệu không được nhìn thấy khi huấn luyện. Một mô hình chỉ mô tả tốt training set là mô hình không có *tính tổng quát* (*generalization*). Một mô hình tốt là mô hình có tính tổng quát.

Để có cái nhìn đầu tiên về overfitting, chúng ta cùng xem Hình 8.1. Có 50 điểm dữ liệu, ở đó đầu ra bằng một đa thức bậc ba của đầu vào cộng thêm nhiễu. Tập dữ liệu này được chia làm hai phần: 30 điểm dữ liệu màu đỏ là training set, 20 điểm dữ liệu màu vàng là dữ liệu kiểm thử. Đồ thị của đa thức bậc ba này được cho bởi đường nét đứt màu xanh lục. Bài toán đặt ra là giả sử ta không biết mô hình ban đầu mà chỉ biết các điểm dữ liệu, hãy tìm một mô hình tốt để mô tả quan hệ giữa đầu vào và đầu ra của dữ liệu đã cho. Giả sử biết thêm rằng mô hình được mô tả bởi một đa thức.

Nhắc lại đa thức nội suy Lagrange. Cho N cặp điểm dữ liệu $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$ với các x_i khác nhau đôi một, luôn tìm được một đa thức $P(\cdot)$ bậc không vượt quá $N - 1$ sao cho $P(x_i) = y_i$, $\forall i = 1, 2, \dots, N$.



Hình 8.1: Underfitting và overfitting với polynomial regression.

Như đã đề cập trong Chương 7, với loại dữ liệu này, chúng ta có thể áp dụng polynomial regression với vector đặc trưng là $\mathbf{x} = [1, x, x^2, x^3, \dots, x^d]^T$ cho đa thức bậc d . Điều quan trọng là cần xác định bậc d của đa thức.

Rõ ràng là một đa thức bậc không vượt quá 29 có thể *fit* được hoàn toàn với 30 điểm trong tập training set. Xét một vài giá trị $d = 2, 4, 8, 16$. Với $d = 2$, mô hình không thực sự tốt vì mô hình dự đoán (*predicted model*) quá khác so với mô hình thực (*true model*); thậm chí nó có xu hướng đi xuống khi mà dữ liệu đang có hướng đi lên. Trong trường hợp này, ta nói mô hình bị *underfitting*. Với $d = 8$, với các điểm dữ liệu trong training set, mô hình dự đoán và mô hình thực là khá giống nhau. Tuy nhiên, về phía phải, đa thức bậc 8 cho kết quả hoàn toàn ngược với xu hướng của dữ liệu. Điều tương tự xảy ra trong trường hợp $d = 16$. Đa thức bậc 16 này *quá fit* training set. Việc *quá fit* trong trường hợp bậc 16 là không tốt vì mô hình có thể đang cố gắng mô tả *nhiều* hơn là dữ liệu. Hiện tượng xảy ra ở hai trường hợp đa thức bậc cao này được gọi là *overfitting*. Độ phức tạp của đô thị trong hai trường hợp này cũng khá lớn, dẫn đến các đường dự đoán không được tự nhiên.

Với $d = 4$, mô hình dự đoán khá giống với mô hình thực. Hệ số bậc cao nhất tìm được rất gần với 0^1 , vì vậy đa thức bậc bốn này khá gần với đa thức bậc ba ban đầu. Đây chính là một mô hình tốt.

Overfitting là hiện tượng mô hình tìm được *quá khớp* với dữ liệu huấn luyện. Việc này sẽ gây ra hậu quả lớn nếu trong training set có nhiễu. Khi đó, mô hình quá chú trọng vào việc xấp xỉ training set mà quên đi việc quan trọng hơn là tính tổng quát, khiến cho mô hình không thực sự mô tả tốt dữ liệu ngoài training set. Overfitting đặc biệt xảy ra khi lượng dữ liệu huấn luyện quá nhỏ hoặc độ phức tạp của mô hình quá cao. Trong ví dụ trên đây, độ phức tạp của mô hình có thể được coi là bậc của đa thức cần tìm.

Vậy, có những kỹ thuật nào giúp tránh overfitting?

Trước hết, chúng ta cần một vài đại lượng để đánh giá chất lượng của mô hình trên training set và test set. Dưới đây là hai đại lượng đơn giản, với giả sử \mathbf{y} là đầu ra thực sự (có thể là vector), và $\hat{\mathbf{y}}$ là đầu ra dự đoán bởi mô hình.

Training error: Đại lượng này là mức độ sai khác giữa đầu ra thực và đầu ra dự đoán của mô hình, thường là giá trị của hàm mất mát áp dụng lên training data. Hàm mất mát này cần có một thừa số $\frac{1}{N_{\text{train}}}$ để tính giá trị trung bình, tức mất mát trung bình trên mỗi điểm dữ liệu. Với các bài toán regression, đại lượng này thường được xác định bởi *mean squared error* (MSE).

$$\text{training error} = \frac{1}{2N_{\text{train}}} \sum_{\text{training set}} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

Với classification, trung bình cộng của cross entropy loss (với softmax regression) hoặc hinge loss (với multi-class SVM) thường được sử dụng.

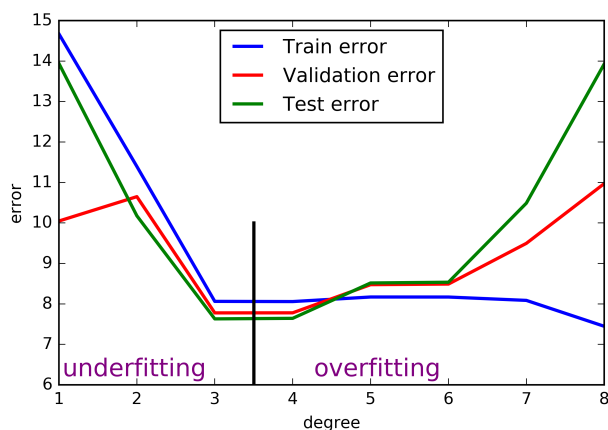
Test error: Tương tự như trên, nhưng mô hình tìm được được áp dụng vào test data. Chú ý rằng, khi xây dựng mô hình, ta không được sử dụng thông tin trong tập dữ liệu này. Với regression, đại lượng này thường được định nghĩa bởi

$$\text{test error} = \frac{1}{2N_{\text{test}}} \sum_{\text{test set}} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$$

Việc lấy trung bình là quan trọng vì lượng dữ liệu trong tập huấn luyện và tập kiểm thử có thể chênh lệch rất nhiều.

Một mô hình được coi là tốt (fit) nếu cả *training error* và *test error* đều thấp. Nếu *training error* thấp nhưng *test error* cao, ta nói mô hình bị overfitting. Nếu *training error* cao và *test error* cao, ta nói mô hình bị underfitting. Xác suất để xảy ra việc *training error* cao nhưng *test error* thấp là rất nhỏ. Trong chương này, chúng ta sẽ làm quen với hai kỹ thuật phổ biến giúp tránh overfitting là *validation* và *regularization*.

¹ Source code tại <https://goo.gl/uD9hm1>.



Hình 8.2: Lựa chọn mô hình dựa trên validation error.

8.2 Validation

8.2.1 Validation

Một mô hình là tốt nếu cả training error và test error đều nhỏ. Tuy nhiên, khi xây dựng một mô hình *chỉ* dựa trên training set, làm thế nào để biết được chất lượng của nó trên test set?

Phương pháp đơn giản nhất là *trích* từ training set ra một tập con nhỏ và thực hiện việc đánh giá mô hình trên tập con nhỏ này. Tập con nhỏ **được trích ra từ training set** này được gọi là *validation set*. Lúc này, **training set mới là phần còn lại của training set ban đầu**. Việc này khá giống với việc chúng ta ôn thi. Giả sử các đề thi của các năm trước là training set, đề thi năm nay là test set mà ta chưa biết. Khi ôn tập, ta thường chia đề các năm trước ra hai phần: phần thứ nhất có thể xem lời giải và tài liệu để ôn tập, phần còn lại ta tự làm mà không sử dụng tài liệu để tự đánh giá kiến thức của mình. Lúc này, phần thứ nhất đóng vai trò là training set mới, trong khi phần thứ hai chính là validation set. Nếu kết quả bài làm trên phần thứ hai là khả quan, ta có thể tự tin hơn khi vào bài thi thật.

Lúc này, có ba đại lượng cần được quan tâm: *training error* trên training set mới, *validation error* được định nghĩa tương tự trên validation set, và *test error* trên test set. Với khái niệm mới này, ta tìm mô hình sao cho cả *train error* và *validation error* đều nhỏ, qua đó có thể dự đoán được rằng *test error* cũng nhỏ. Để làm được việc này, ta có thể huấn luyện nhiều mô hình khác nhau dựa trên training set, sau đó áp dụng các mô hình tìm được và tính *validation error*. Mô hình cho *validation error* nhỏ nhất sẽ là một mô hình tốt.

Thông thường, ta bắt đầu từ mô hình đơn giản, sau đó tăng dần độ phức tạp của mô hình. Khi độ phức tạp này tăng lên, training error sẽ có xu hướng nhỏ dần, nhưng điều tương tự có thể không xảy ra ở validation error. Validation error ban đầu thường giảm dần và đến một lúc sẽ tăng lên do overfitting xảy ra trên training khi độ phức tạp của mô hình tăng lên. Để chọn ra một mô hình tốt, ta quan sát validation error. Khi *validation error* có chiều hướng tăng lên thì ta chọn mô hình trước đó.

Hình 8.2 mô tả ví dụ phía trên với bậc của đa thức tăng từ một đến tám. Validation set là 10 điểm được lấy ra ngẫu nhiên từ training set 30 điểm ban đầu. Chúng ta hãy tạm chỉ xét

hai đường màu lam và đỏ, tương ứng với training error và validation error. Khi bậc của đa thức tăng lên, training error có xu hướng giảm. Điều này dễ hiểu vì đa thức bậc càng cao, việc xấp xỉ càng chính xác. Quan sát đường màu đỏ, khi bậc của đa thức là ba hoặc bốn thì validation error thấp, sau đó nó tăng dần lên. Dựa vào validation error, ta có thể xác định được bậc cần chọn là ba hoặc bốn. Quan sát tiếp đường màu lục, tương ứng với test error, thật là trùng hợp, với bậc bằng ba hoặc bốn, test error cũng đạt giá trị nhỏ nhất, sau đó tăng dần lên. Vậy cách làm này ở đây đã tỏ ra hiệu quả. Và mô hình phù hợp là mô hình có bậc bằng ba hoặc bốn. Trong ví dụ này, validation set đóng vai trò tìm ra bậc của đa thức, training set đóng vai trò trong việc tìm các hệ số của đa thức với bậc đã biết. Các hệ số của đa thức chính là các *model parameter*, trong khi bậc của đa thức có thể được coi là *hyperparameter*. Cả training set và validation set đều đóng vai trò xây dựng mô hình. Nhắc lại rằng hai tập hợp này được tách ra từ training set ban đầu.

Việc không sử dụng test data khi lựa chọn mô hình ở trên nhưng vẫn có được kết quả khả quan vì ta đã giả sử rằng validation data và test data có chung một đặc điểm nào đó (chung phân phối và đều chưa được mô hình nhìn thấy khi huấn luyện). Và khi cả hai đều là chưa được nhìn thấy, *error* trên hai tập này sẽ tương đối giống nhau.

Để ý rằng, khi bậc nhỏ (bằng một hoặc hai), cả ba error đều cao, khi đó *underfitting* xảy ra.

8.2.2 Cross-validation

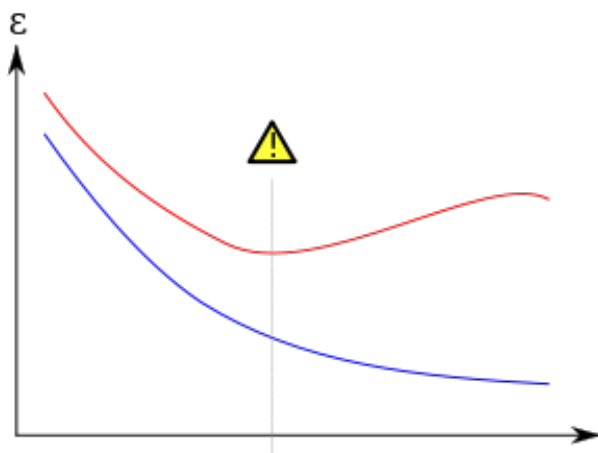
Trong nhiều trường hợp, chúng ta có rất hạn chế số lượng dữ liệu để xây dựng mô hình. Nếu lấy quá nhiều dữ liệu trong training set ra làm dữ liệu validation, phần dữ liệu còn lại của training set là không đủ để xây dựng mô hình. Lúc này, validation set phải thật nhỏ để giữ được lượng dữ liệu cho training đủ lớn. Tuy nhiên, một vấn đề khác nảy sinh. Khi validation set quá nhỏ, hiện tượng overfitting lại có thể xảy ra với training set còn lại. Có giải pháp nào cho tình huống này không?

Câu trả lời là *cross-validation*.

Cross-validation là một cải tiến của validation với lượng dữ liệu trong validation set là nhỏ nhưng chất lượng mô hình được đánh giá trên nhiều tập validation khác nhau. Một cách thường được sử dụng là chia training set ra k tập con không giao nhau, có kích thước gần bằng nhau. Tại mỗi lần, được gọi là một *run*, một trong số k tập con được lấy ra làm validation set. Nhiều mô hình khác nhau sẽ được xây dựng dựa vào hợp của $k - 1$ tập con còn lại. Mô hình cuối được xác định dựa trên trung bình của các training error và validation error. Cách làm này còn có tên gọi là **k-fold cross-validation**.

Khi k bằng với số lượng phần tử trong training set ban đầu, tức mỗi tập con có đúng một phần tử, ta gọi kỹ thuật này là **leave-one-out**.

Thư viện scikit-learn hỗ trợ rất nhiều phương thức cho phân chia dữ liệu và tính toán error của các mô hình. Bạn đọc có thể xem thêm *Cross-validation: evaluating estimator performance* (<https://goo.gl/Ars2cr>).



Hình 8.3: Early stopping. Đường màu xanh là training error, màu đỏ là validation error. Trục x thể hiện số lượng vòng lặp, trục y là giá trị error. Thuật toán huấn luyện dừng lại tại vòng lặp mà validation error đạt giá trị nhỏ nhất (Nguồn: [Overfitting – Wikipedia](#)).

8.3 Regularization

Một nhược điểm lớn của *cross-validation* là số lượng mô hình cần huấn luyện tỉ lệ thuận với k . Điều đáng nói là mô hình polynomial regression như trên chỉ có một tham số liên quan đến độ phức tạp của mô hình cần xác định là bậc của đa thức. Trong các nhiều bài toán, lượng tham số cần xác định thường lớn hơn nhiều, và khoảng giá trị của mỗi tham số cũng rộng hơn nhiều, chưa kể đến việc có những tham số có thể là số thực. Điều này dẫn đến việc huấn luyện nhiều mô hình là khó khả thi. Có một kỹ thuật giúp số mô hình cần huấn luyện giảm đi nhiều, thậm chí chỉ một mô hình. Kỹ thuật này có tên gọi là *regularization*.

Regularization, một cách dễ hiểu, là thay đổi mô hình một chút, chấp nhận hy sinh độ chính xác trong training set, nhưng giảm độ phức tạp của mô hình, giúp tránh overfitting trong khi vẫn giữ được tính tổng quát của nó. Dưới đây là một vài kỹ thuật regularization.

8.3.1 Early stopping

Rất nhiều các mô hình machine learning được xây dựng bằng cách sử dụng các thuật toán lặp cho tới khi hàm mất mát hội tụ để tìm ra nghiệm. Nhìn chung, giá trị hàm mất mát giảm dần khi số vòng lặp tăng lên. Một giải pháp giúp giảm overfitting là dừng thuật toán trước khi nó hội tụ. Giải pháp này có tên là *early stopping*.

Vậy dừng khi nào là phù hợp? Một kỹ thuật thường được sử dụng là tách từ training set ra một validation set như trên. Trong khi huấn luyện, ta tính toán cả training error và validation error, nếu training error vẫn có xu hướng giảm nhưng validation error có xu hướng tăng lên thì ta dừng thuật toán.

Hình 8.3 mô tả cách tìm điểm *stopping*. Chúng ta thấy rằng phương pháp này khá giống với phương pháp tìm bậc của đa thức ở phần trên của bài viết, với độ phức tạp của mô hình có thể được coi là số vòng lặp cần chạy. Số vòng lặp càng cao thì hàm mất mát càng nhỏ, tức mô hình có khả năng *quá fit* với training set.

8.3.2 Thêm số hạng vào hàm mất mát

Kỹ thuật regularization phổ biến hơn là thêm vào hàm mất mát một số hạng nữa. Số hạng này thường dùng để đánh giá độ phức tạp của mô hình với giá trị lớn thể hiện mô hình phức tạp. *Hàm mất mát mới* này được gọi là **regularized loss function**, thường được định nghĩa như sau:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \lambda R(\theta)$$

Nhắc lại rằng θ được dùng để ký hiệu các tham số trong mô hình. $\mathcal{L}(\theta)$ là hàm mất mát phụ thuộc vào training set và θ , $R(\theta)$ là số hạng *regularization* chỉ phụ thuộc vào θ . Số vô hướng λ thường là một số dương nhỏ, còn được gọi là *tham số regularization (regularization parameter)*. Tham số regularization thường được chọn là các giá trị nhỏ để đảm bảo nghiệm của bài toán tối ưu $\mathcal{L}_{\text{reg}}(\theta)$ không quá xa nghiệm của bài toán tối ưu $\mathcal{L}(\theta)$.

Hai hàm regularization phổ biến là ℓ_1 norm và ℓ_2 norm regularization (viết gọn là ℓ_1 regularization và ℓ_2 regularization). Ví dụ, khi chọn $R(\mathbf{w}) = \|\mathbf{w}\|_2^2$ cho hàm mất mát của linear regression, chúng ta sẽ đạt được ridge regression. Hàm regularization này khiến các hệ số trong \mathbf{w} không quá lớn, giúp tránh việc đầu ra phụ thuộc quá nhiều vào một đặc trưng nào đó. Trong khi đó, khi chọn $R(\mathbf{w}) = \|\mathbf{w}\|_1$, nghiệm \mathbf{w} tìm được có xu hướng rất nhiều phần tử bằng không (*sparse solution*²). Khi thêm ℓ_1 regularization vào hàm mất mát của linear regression, chúng ta sẽ thu được LASSO regression. Các thành phần khác không của \mathbf{w} tương đương với các đặc trưng quan trọng đóng góp vào việc dự đoán đầu ra. Các đặc trưng ứng với thành phần bằng không của \mathbf{w} được coi là ít quan trọng. Chính vì vậy, LASSO regression cũng được coi là một phương pháp giúp lựa chọn những đặc trưng hữu ích cho mô hình; nó cũng là một trong các phương pháp *feature selection*.

ℓ_1 regularization được cho là giúp cho mô hình *robust* (ít bị ảnh hưởng bởi nhiễu) hơn so với ℓ_2 regularization. Tuy nhiên, hạn chế của ℓ_1 regularization là đạo hàm của ℓ_1 norm không xác định tại không (đạo hàm của hàm trị tuyệt đối), dẫn đến việc tìm nghiệm thường tốn thời gian hơn. Trong khi đó, đạo hàm của ℓ_2 norm xác định mọi nơi, và trong nhiều trường hợp, ta có thể tìm được công thức nghiệm cho phương trình đạo hàm của (regularized) loss function bằng không. Các nghiệm có công thức xác định được gọi là *closed-form solution*.

Trong neural network, việc sử dụng ℓ_2 regularization còn được gọi là *weight decay* [KH92]. Ngoài ra, gần đây một phương pháp regularization rất hiệu quả cho neural network được sử dụng là *dropout* [SHK⁺14].

8.4 Đọc thêm

1. A. Krogh *et al.*, *A simple weight decay can improve generalization*. NIPS 1991 [KH92].
2. N. Srivastava *et al.*, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15.1 (2014): 1929-1958 [SHK⁺14].
3. *Understanding the Bias-Variance Tradeoff* (<https://goo.gl/yvQv3w>).

² *L1 Norm Regularization and Sparsity Explained for Dummies* (<https://goo.gl/VqPTLh>).

Phần III

Khởi động

Trong phần này, chúng ta sẽ làm quen với ba thuật toán machine learning chưa cần nhiều tối ưu: k-means clustering cho phân nhóm dữ liệu, k-nearest neighbors cho classification và regression, naive Bayes classifier cho phân loại văn bản.

K-nearest neighbors

Nếu như con người có kiểu học “nước đến chân mới nhảy” thì machine learning cũng có một thuật toán như vậy.

9.1 Giới thiệu

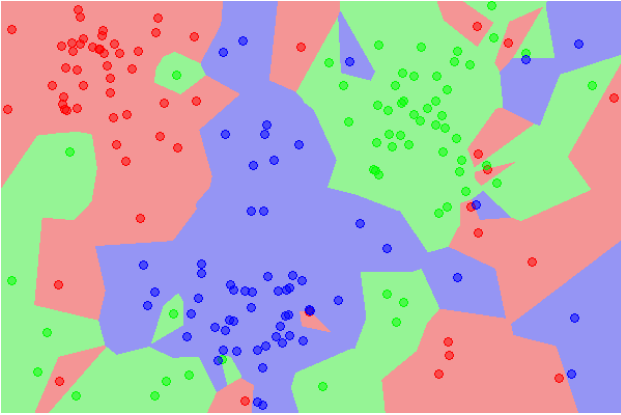
9.1.1 K-nearest neighbor

K-nearest neighbor (KNN) là một trong những thuật toán supervised learning đơn giản. Khi huấn luyện, thuật toán này *không học* một điều gì từ dữ liệu huấn luyện mà *nhớ* lại một cách máy móc toàn bộ dữ liệu đó. Đây cũng là lý do thuật toán này được xếp vào loại *lazy learning*, mọi tính toán được thực hiện khi nó cần dự đoán đầu ra của dữ liệu mới. KNN có thể áp dụng được vào cả classification và regression. KNN còn được gọi là một thuật toán *instance-based* [AKA91] hay memory-based learning.

Với KNN, trong bài toán classification, nhãn của một điểm dữ liệu mới được suy ra trực tiếp từ K điểm dữ liệu gần nhất trong tập huấn luyện. Nhãn đó có thể được quyết định bằng *bầu chọn theo đa số* (*major voting*) trong số K điểm gần nhất, hoặc nó có thể được suy ra bằng cách đánh trọng số khác nhau cho mỗi trong các điểm gần nhất đó rồi suy ra kết quả. Chi tiết sẽ được nêu trong phần tiếp theo. Trong bài toán regression, đầu ra của một điểm dữ liệu sẽ bằng chính đầu ra của điểm dữ liệu đã biết gần nhất (trong trường hợp $K = 1$), hoặc là trung bình có trọng số của đầu ra của những điểm gần nhất, hoặc bằng một mối quan hệ dựa trên các điểm gần nhất đó và khoảng cách tới chúng.

Một cách ngắn gọn, KNN là thuật toán đi tìm đầu ra của một điểm dữ liệu mới bằng cách chỉ dựa trên thông tin của K điểm dữ liệu gần nhất trong tập huấn luyện.

Hình 9.1 mô tả một bài toán phân lớp với ba class: đỏ, lam, lục. Các hình tròn nhỏ với màu khác nhau thể hiện dữ liệu huấn luyện của các class khác nhau. Các vùng màu nền khác nhau thể hiện *lãnh thổ* của mỗi class. Tại một điểm bất kỳ, class của nó được xác định dựa trên class của điểm gần nó nhất trong tập huấn luyện. Trong hình này, có một vài



Hình 9.1: Ví dụ về 1NN. Các hình tròn là các điểm dữ liệu huấn luyện. Các hình khác màu thể hiện các lớp khác nhau. Các vùng nền thể hiện các điểm được phân loại vào lớp có màu tương ứng khi sử dụng 1NN (Nguồn: [K-nearest neighbors algorithm – Wikipedia](#)).

vùng nhỏ xem lẫn vào các vùng lớn hơn khác màu. Ví dụ có một điểm màu lục ở gần góc 11 giờ nằm giữa hai vùng lớn với nhiều dữ liệu màu đỏ và lam. Điểm này rất có thể là nhiễu. Việc này nhiều khả năng sẽ dẫn đến việc phân lớp sai cho một điểm dữ liệu kiểm thử rơi vào khu vực này.

KNN là một ví dụ rõ nhất của overfitting. Với mô hình này, mọi điểm trong tập huấn luyện đều được mô hình mô tả một cách chính xác, vì vậy, nó rất nhạy cảm với nhiễu.

Mặc dù có nhiều hạn chế, KNN vẫn là một giải pháp đầu tiên nên nghĩ tới khi giải quyết một bài toán machine learning. *Khi làm các bài toán machine learning nói chung, không có mô hình đúng hay sai, chỉ có mô hình cho kết quả tốt hơn. Chúng ta luôn cần một mô hình đơn giản để giải quyết bài toán, sau đó dần dần tìm cách tăng chất lượng của mô hình.*

9.2 Phân tích toán học

KNN là một thuật toán *lazy learning*, không có hàm mất mát nào và bài toán tối ưu nào phải thực hiện trong quá trình huấn luyện. Mọi tính toán được thực hiện ở bước kiểm thử. Vì KNN ra quyết định dựa trên các điểm gần nhất nên có hai vấn đề ta cần lưu tâm. Thứ nhất, khoảng cách được định nghĩa như thế nào. Thứ hai, cần phải tính toán khoảng cách như thế nào cho hiệu quả.

Với vấn đề thứ nhất, mỗi điểm dữ liệu được thể hiện bằng một vector đặc trưng, khoảng cách giữa hai điểm chính là khoảng cách giữa hai vector đó. Để đo khoảng cách trong không gian nhiều chiều, norm (xem Mục 1.14) thường được sử dụng, và norm phổ biến nhất là ℓ_2 norm, chính là khoảng cách Euclid quen thuộc.

Ta cần lưu ý tới vấn đề thứ hai hơn, đặc biệt với các bài toán với tập huấn luyện lớn và vector đặc trưng có kích thước lớn (*large-scale problem*). Giả sử các vector đặc trưng huấn luyện là các cột của ma trận $\mathbf{X} \in \mathbb{R}^{d \times N}$ với d và N lớn, KNN sẽ phải tính toán khoảng cách từ một điểm dữ liệu mới $\mathbf{z} \in \mathbb{R}^d$ đến toàn bộ N điểm dữ liệu đã cho và chọn ra K khoảng cách nhỏ nhất. Nếu không có một cách tính hiệu quả, khối lượng tính toán ở đây sẽ rất lớn. Đây cũng là cái giá phải trả cho việc *lazy learning*.

Tiếp theo, chúng ta cùng thực hiện một vài phân tích toán học để tìm ra cách tính các khoảng cách một cách hiệu quả.

Khoảng cách từ một điểm tới từng điểm trong một tập hợp

Khoảng cách Euclid từ một điểm \mathbf{z} tới một điểm \mathbf{x}_i trong tập huấn luyện được định nghĩa bởi $\|\mathbf{z} - \mathbf{x}_i\|_2$. Vì trong cách tính này có một bước phải tính căn bậc hai nên người ta thường tính $\|\mathbf{z} - \mathbf{x}_i\|_2^2$. Nếu việc tính khoảng cách chỉ để phục vụ việc sắp xếp thì ta không cần tính căn bậc hai sau bước này nữa. Để ý rằng

$$\|\mathbf{z} - \mathbf{x}_i\|_2^2 = (\mathbf{z} - \mathbf{x}_i)^T (\mathbf{z} - \mathbf{x}_i) = \|\mathbf{z}\|_2^2 + \|\mathbf{x}_i\|_2^2 - 2\mathbf{x}_i^T \mathbf{z} \quad (9.1)$$

Từ đây, nếu nhiệm vụ chỉ là tìm ra \mathbf{x}_i gần với \mathbf{z} nhất, số hạng đầu tiên có thể được bỏ qua. Hơn nữa, nếu có nhiều điểm dữ liệu trong tập kiểm thử, các giá trị $\|\mathbf{x}_i\|_2^2$ có thể được tính và lưu trước vào bộ nhớ. Khi đó, ta chỉ cần tính các tích vô hướng $\mathbf{x}_i^T \mathbf{z}$.

Để thấy rõ hơn, chúng ta cùng làm một ví dụ trên Python. Trước hết, chọn d và N là các giá trị lớn và khai báo ngẫu nhiên \mathbf{X} và \mathbf{z} . Lưu ý rằng vì dữ liệu trong Python thường được lưu ở dạng hàng, khi lập trình ta cần thay đổi một chút.

```
from __future__ import print_function
import numpy as np
from time import time # for comparing runing time
d, N = 1000, 10000 # dimension, number of training points
X = np.random.randn(N, d) # N d-dimensional points
z = np.random.randn(d)
```

Tiếp theo, ta viết ba hàm số:

1. `dist_pp(z, x)` tính bình phương khoảng cách Euclid giữa hai vector \mathbf{z} và \mathbf{x} . Hàm này thực hiện cách tính trực tiếp, tức tính hiệu rồi lấy bình phương ℓ_2 norm của vector hiệu.
2. `dist_ps_naive(z, X)` tính bình phương khoảng cách giữa \mathbf{z} và mỗi hàng của \mathbf{X} . Trong hàm này, các khoảng cách được tính dựa trên việc tính từng cặp `dist_pp(z, X[i])`.
3. `dist_ps_fast(z, X)` tính bình phương khoảng cách giữa \mathbf{z} và mỗi hàng của \mathbf{X} , tuy nhiên, kết quả được tính dựa trên đẳng thức (9.1). Khi có nhiều điểm dữ liệu được lưu trong \mathbf{X} , ta cần tính tổng bình phương các phần tử của mỗi điểm dữ liệu và tính tích $\mathbf{X}^T \mathbf{z}$.

Đoạn code dưới đây thể hiện cách tính khoảng cách từ một điểm \mathbf{z} tới một tập hợp điểm \mathbf{X} bằng hai cách. Kết quả và thời gian chạy của mỗi hàm cũng được in ra để so sánh.

```

# naively compute square distance between two vector
def dist_pp(z, x):
    d = z - x.reshape(z.shape) # force x and z to have the same dims
    return np.sum(d*d)

# from one point to each point in a set, naive
def dist_ps_naive(z, X):
    N = X.shape[0]
    res = np.zeros((1, N))
    for i in range(N):
        res[0][i] = dist_pp(z, X[i])
    return res

# from one point to each point in a set, fast
def dist_ps_fast(z, X):
    X2 = np.sum(X*X, 1) # square of l2 norm of each ROW of X
    z2 = np.sum(z*z) # square of l2 norm of z
    return X2 + z2 - 2*X.dot(z) # z2 can be ignored

t1 = time()
D1 = dist_ps_naive(z, X)
print('naive point2set, running time:', time() - t1, 's')

t1 = time()
D2 = dist_ps_fast(z, X)
print('fast point2set , running time:', time() - t1, 's')
print('Result difference:', np.linalg.norm(D1 - D2))

```

Kết quả:

```

naive point2set, running time: 0.0932548046112 s
fast point2set , running time: 0.0514178276062 s
Result difference: 2.11481965531e-11

```

Kết quả chỉ ra rằng hàm `dist_ps_fast(z, X)` chạy nhanh hơn gần gấp đôi so với hàm `dist_ps_naive(z, X)` (số này còn lớn hơn nữa khi kích thước dữ liệu tăng lên). Quan trọng hơn, sự chênh lệch của kết quả hai phép tính là một số rất nhỏ. Điều này chỉ ra rằng `dist_ps_fast(z, X)` nên được ưu tiên hơn.

Khoảng cách giữa từng cặp điểm trong hai tập hợp

Thông thường, ta không những phải tính khoảng cách từ một điểm \mathbf{z} tới tập hợp các điểm \mathbf{X} , mà thường xuyên còn phải tính khoảng cách giữa nhiều điểm \mathbf{Z} tới \mathbf{X} . Nói đúng hơn, ta phải tính từng cặp khoảng cách giữa mỗi điểm trong tập kiểm thử và một điểm trong tập huấn luyện. Nếu mỗi tập có 1000 phần tử, ta sẽ phải tính một triệu khoảng cách—là một số rất lớn. Nếu không có một phương pháp tính hiệu quả, thời gian thực hiện sẽ là rất lớn.

Dưới đây là đoạn code thực hiện cách tính bình phương khoảng cách giữa các cặp điểm trong hai tập điểm sử dụng hai phương pháp khác nhau. Phương pháp thứ nhất sử dụng một vòng `for` tính khoảng cách từ từng điểm trong tập thứ nhất đến tất cả các điểm trong tập thứ

hai sử dụng hàm `dist_ps_fast(z, X)` ở trên. Phương pháp thứ hai tiếp tục sử dụng (9.1) cho trường hợp tổng quát.

```
M = 100
Z = np.random.randn(M, d)

# from each point in one set to each point in another set, half fast
def dist_ss_0(Z, X):
    M = Z.shape[0]
    N = X.shape[0]
    res = np.zeros((M, N))
    for i in range(M):
        res[i] = dist_ps_fast(Z[i], X)
    return res

# from each point in one set to each point in another set, fast
def dist_ss_fast(Z, X):
    X2 = np.sum(X*X, 1) # square of l2 norm of each ROW of X
    Z2 = np.sum(Z*Z, 1) # square of l2 norm of each ROW of Z
    return Z2.reshape(-1, 1) + X2.reshape(1, -1) - 2*Z.dot(X.T)

t1 = time()
D3 = dist_ss_0(Z, X)
print('half fast set2set running time:', time() - t1, 's')
t1 = time()
D4 = dist_ss_fast(Z, X)
print('fast set2set running time', time() - t1, 's')
print('Result difference:', np.linalg.norm(D3 - D4))
```

Kết quả:

```
half fast set2set running time: 4.33642292023 s
fast set2set running time 0.0583250522614 s
Result difference: 9.93586539607e-11
```

Điều này chỉ ra rằng hai cách tính cho kết quả chênh lệch nhau không đáng kể. Trong khi đó `dist_ss_fast(Z, X)` chạy nhanh hơn `dist_ss_0(Z, X)` nhiều lần.

Khi làm việc trên python, chúng ta có thể sử dụng hàm `cdist` (<https://goo.gl/vYMnmM>) trong `scipy.spatial.distance`, hoặc hàm `pairwise_distances` (<https://goo.gl/QK6Zyi>) trong `sklearn.metrics.pairwise`. Các hàm này giúp tính khoảng cách từng cặp điểm trong hai tập hợp khá hiệu quả. Phần còn lại của chương này sẽ trực tiếp sử dụng thư viện scikit-learn cho KNN. Việc viết lại thuật toán này không quá phức tạp khi đã có một hàm tính khoảng cách hiệu quả.

Nếu thực hiện trên GPU, bạn đọc có thể tham khảo thêm bài báo [JDJ17] và source code của nó tại <https://github.com/facebookresearch/faiss>.



Iris setosa

Iris versicolor

Iris virginica

Hình 9.2: Ba loại hoa lan trong bộ cơ sở dữ liệu hoa Iris.

9.3 Ví dụ trên cơ sở dữ liệu Iris

9.3.1 Bộ cơ sở dữ liệu hoa Iris (Iris flower dataset).

Iris flower dataset (<https://goo.gl/eUy83R>) là một bộ dữ liệu nhỏ. Bộ dữ liệu này bao gồm thông tin của ba class hoa Iris (một loài hoa lan) khác nhau: Iris setosa, Iris virginica và Iris versicolor. Mỗi class có 50 bông hoa với dữ liệu là bốn thông tin: chiều dài, chiều rộng đài hoa (sepal), và chiều dài, chiều rộng cánh hoa (petal). Hình 9.2 là ví dụ về hình ảnh của ba loại hoa. Chú ý rằng các điểm dữ liệu không phải là các bức ảnh mà chỉ là một vector đặc trưng bốn chiều gồm bốn thông tin ở trên.

9.3.2 Thí nghiệm

Trong phần này, chúng ta sẽ tách 150 điểm dữ liệu trong Iris flower dataset ra thành hai tập huấn luyện và kiểm thử. KNN sẽ dựa vào thông tin trong tập huấn luyện để dự đoán xem mỗi dữ liệu trong tập kiểm thử tương ứng với loại hoa nào. Dữ liệu được dự đoán này sẽ được đối chiếu với dữ liệu thật để đánh giá hiệu quả của KNN.

Trước tiên, chúng ta cần khai báo vài thư viện. Iris flower dataset có sẵn trong thư viện scikit-learn.

```
from __future__ import print_function
import numpy as np
from sklearn import neighbors, datasets
from sklearn.model_selection import train_test_split # for splitting data
from sklearn.metrics import accuracy_score         # for evaluating results
```

Tiếp theo, ta sẽ load cơ sở dữ liệu này và chọn ra ngẫu nhiên 130 mẫu làm test set, 20 mẫu còn lại được dùng làm training set.

```
np.random.seed(7)
iris = datasets.load_iris()
iris_X = iris.data
iris_y = iris.target
print('Labels:', np.unique(iris_y))

# split train and test
X_train, X_test, y_train, y_test = train_test_split(iris_X, iris_y, test_size=130)
print('Train size:', X_train.shape[0], ', test size:', X_test.shape[0])
```

```
Labels: [0 1 2]
Train size: 20 , test size: 130
```

Dòng `np.random.seed(7)` để đảm bảo rằng khi các bạn chạy lại các đoạn code này cũng nhận được kết quả tương tự. Có thể thay 7 bằng một số tự nhiên bất kỳ 32 bit.

Kết quả với 1NN

Tối đây, ta trực tiếp sử dụng thư viện scikit-learn cho KNN. Xét ví dụ đầu tiên với $K = 1$.

```
model = neighbors.KNeighborsClassifier(n_neighbors = 1, p = 2)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy of 1NN: %.2f %" % (100*accuracy_score(y_test, y_pred)))
```

Kết quả:

```
Accuracy of 1NN: 92.31 %
```

Kết quả thu được là 92.31% (tỉ lệ các mẫu được phân loại chính xác). Ở đây, `n_neighbors = 1` chỉ ra rằng ta chỉ lấy một điểm gần nhất, tức $K = 1$, `p = 2` chính là l_2 norm ta đã chọn để tính khoảng cách. Bạn đọc có thể thử với `p = 1` tương ứng với khoảng cách l_1 norm.

Kết quả với 7NN

Như đã đề cập, 1NN rất dễ gây ra hiện tượng overfitting. Để giảm thiểu việc này, ta có thể tăng lượng điểm lân cận lên, ví dụ bảy điểm, và xem xét trong bảy điểm gần nhất, đa số chúng thuộc vào class nào.

```
model = neighbors.KNeighborsClassifier(n_neighbors = 7, p = 2)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy of 7NN with major voting: %.2f %" % (100*accuracy_score(y_test,
% y_pred)))
```

Kết quả:

```
Accuracy of 7NN with major voting: 93.85 %
```

Ta nhận thấy rằng độ chính xác đã tăng lên khi ta dự đoán dựa trên nhiều lân cận hơn.

Đánh trọng số cho các điểm lân cận

Trong kỹ thuật major voting bên trên, mỗi trong bảy điểm gần nhất được coi là có vai trò như nhau và giá trị *lá phiếu* của mỗi điểm này là như nhau. Như thế có thể không công bằng, vì những điểm gần hơn cần có trọng số cao hơn. Vì vậy, ta sẽ đánh trọng số khác nhau cho mỗi trong bảy điểm gần nhất này. Cách đánh trọng số phải thoả mãn điều kiện là một điểm càng gần điểm kiểm thử phải được đánh trọng số càng cao. Cách đơn giản nhất là lấy nghịch đảo của khoảng cách này. *Trong trường hợp test data trùng với một điểm dữ liệu trong training data, tức khoảng cách bằng 0, ta lấy luôn đầu ra của điểm training data.*

Scikit-learn giúp chúng ta đơn giản hóa việc này bằng cách gán thuộc tính `weights = 'distance'`. (Giá trị mặc định của `weights` là `'uniform'`, tương ứng với việc coi tất cả các điểm lân cận có giá trị như nhau như ở trên).

```
model = neighbors.KNeighborsClassifier(n_neighbors = 7, p = 2, weights = 'distance')
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy of 7NN (1/distance weights): %.2f %" % (100*accuracy_score(y_test,
    y_pred)))
```

Kết quả:

```
Accuracy of 7NN (1/distance weights): 94.62 %
```

Độ chính xác tiếp tục được tăng lên.

K-nearest neighbors với trọng số tự định nghĩa

Ngoài 2 phương pháp đánh trọng số `weights = 'uniform'` và `weights = 'distance'` ở trên, scikit-learn còn cung cấp cho chúng ta một cách để đánh trọng số một cách tùy chọn. Ví dụ, một cách đánh trọng số phổ biến khác thường được dùng là

$$w_i = \exp\left(\frac{-\|\mathbf{z} - \mathbf{x}_i\|_2^2}{\sigma^2}\right)$$

trong đó w_i là trọng số của điểm gần thứ i (\mathbf{x}_i) của điểm dữ liệu đang xét \mathbf{z} , σ là một số dương. Hàm số này cũng thoả mãn điều kiện điểm càng gần \mathbf{x} thì trọng số càng cao (cao nhất bằng 1). Với hàm số này, chúng ta có thể lập trình như sau:

```
def myweight(distances):
    sigma2 = .4 # we can change this number
    return np.exp(-distances**2/sigma2)

model = neighbors.KNeighborsClassifier(n_neighbors = 7, p = 2, weights = myweight)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy of 7NN (customized weights): %.2f %%"
      (100*accuracy_score(y_test, y_pred)))
```

Kết quả:

```
Accuracy of 7NN (customized weights): 95.38 %
```

Kết quả tiếp tục tăng lên một chút. Với từng bài toán, chúng ta có thể thay các thuộc tính của KNN bằng các giá trị khác nhau và chọn ra giá trị tốt nhất thông qua cross-validation.

9.4 Thảo luận

9.4.1 KNN cho Regression

Với bài toán regression, chúng ta cũng hoàn toàn có thể sử dụng phương pháp tương tự: đầu ra của một điểm được xác định dựa trên đầu ra của các điểm lân cận và khoảng cách tới chúng. Giả sử $\mathbf{x}_1, \dots, \mathbf{x}_K$ là K điểm lân cận của một điểm dữ liệu \mathbf{z} với đầu ra tương ứng là y_1, \dots, y_K . Giả sử các trọng số ứng với các lân cận này tính được là w_1, \dots, w_K . Kết quả dự đoán đầu ra của \mathbf{z} có thể được xác định bởi

$$\frac{w_1 y_1 + w_2 y_2 + \dots + w_K y_K}{w_1 + w_2 + \dots + w_K} \quad (9.2)$$

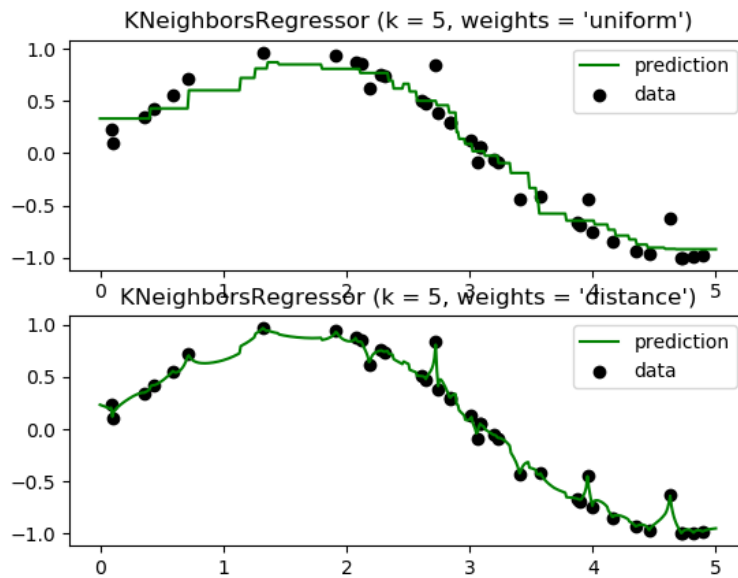
Hình 9.3 là một ví dụ về KNN cho regression với $K = 5$, sử dụng hai cách đánh trọng số khác nhau. Ta có thể thấy rằng `weights = 'distance'` có xu hướng gây ra overfitting.

9.4.2 Ưu điểm của KNN

1. Độ phức tạp tính toán của quá trình huấn luyện là bằng 0.
2. Việc dự đoán kết quả của dữ liệu mới rất đơn giản (sau khi đã xác định được các điểm lân cận).
3. Không cần giả sử về phân phối của các class.

9.4.3 Nhược điểm của KNN

1. KNN rất nhạy cảm với nhiễu khi K nhỏ.



Hình 9.3: KNN cho bài toán Regression (Nguồn: *Nearest neighbors regression–scikit-learn–<https://goo.gl/9VyBF3>*).

- Như đã nói, KNN là một thuật toán mà mọi tính toán đều nằm ở khâu kiểm thử. Trong đó việc tính khoảng cách tới từng điểm dữ liệu trong tập huấn luyện tốn rất nhiều thời gian, đặc biệt là với các cơ sở dữ liệu có số chiều lớn và có nhiều điểm dữ liệu. Với K càng lớn thì độ phức tạp cũng sẽ tăng lên. Ngoài ra, việc lưu toàn bộ dữ liệu trong bộ nhớ cũng ảnh hưởng tới hiệu năng của KNN.

9.4.4 Đọc thêm

- Tutorial To Implement k-Nearest Neighbors in Python From Scratch* (<https://goo.gl/J78Qso>).
- Source code cho chương này có thể được tìm thấy tại <https://goo.gl/asF58Q>.

K-means clustering

10.1 Giới thiệu

Chúng ta đã làm quen với linear regression, một mô hình đơn giản trong supervised learning. Trong chương này, một mô hình đơn giản khác trong unsupervised learning sẽ được trình bày. Thuật toán này có tên là *phân nhóm K-means* (*K-means clustering*).

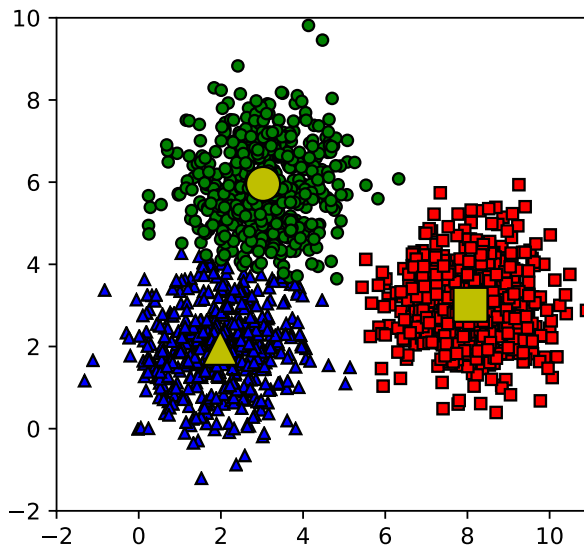
Trong *K-means clustering*, chúng ta không biết nhãn của từng điểm dữ liệu. Mục đích là làm thế nào để phân dữ liệu thành các cụm (cluster) khác nhau sao cho dữ liệu trong cùng một cụm có những tính chất giống nhau.

Ví dụ: Một công ty muốn tạo ra một chính sách ưu đãi cho những nhóm khách hàng khác nhau dựa trên sự tương tác giữa mỗi khách hàng với công ty đó (số năm là khách hàng; số tiền khách hàng đã chi trả cho công ty; độ tuổi; giới tính; thành phố; nghề nghiệp; v.v.). Giả sử công ty đó có rất nhiều dữ liệu của rất nhiều khách hàng nhưng chưa làm công việc phân nhóm khách hàng. *K-means clustering* là một thuật toán có thể giúp thực hiện công việc này. Sau khi đã phân ra được từng nhóm, nhân viên công ty đó có thể lựa chọn ra một vài khách hàng trong mỗi nhóm để quyết định xem mỗi nhóm tương ứng với nhóm khách hàng nào. Phần việc cuối cùng này cần sự can thiệp của con người, nhưng lượng công việc đã được rút gọn đi rất nhiều.

Một định nghĩa đơn giản của nhóm/cụm (*cluster*) là tập hợp các điểm có các vector đặc trưng *gần* nhau. Việc đo khoảng cách giữa các vector thường được thực hiện dựa trên norm, trong đó khoảng cách Euclid, tức l_2 norm, được sử dụng phổ biến hơn cả.

Hình 10.1 là một ví dụ về dữ liệu được phân vào ba cluster¹. Giả sử mỗi cluster có một điểm đại diện (*centroid*) màu vàng, và nhóm của mỗi điểm được xác định qua việc nó gần với centroid nào nhất trong ba centroid. Tới đây, chúng ta có một bài toán thú vị: *Trên một vùng*

¹ Để cho thống nhất, từ *cluster* sẽ được sử dụng thay thế cho nhóm/cụm. Các thuật ngữ tiếng Anh từ đây cũng được sử dụng thường xuyên hơn.



Hình 10.1: Ví dụ với ba cụm dữ liệu trong không gian hai chiều.

biểu hình vuông lớn có ba đảo hình vuông, tam giác, và tròn màu vàng như tròn Hình 10.1. Một điểm trên biển được gọi là thuộc lãnh hải của một đảo nếu nó nằm gần đảo này hơn so với hai đảo kia. Hãy xác định ranh giới lãnh hải giữa các đảo.

Cũng trên Hình 10.1, các vùng với màu nền khác nhau biểu diễn lãnh hải của mỗi đảo. Chúng ta thấy rằng đường phân định giữa các lãnh hải là các đường thẳng. Chính xác hơn, chúng là các đường trung trực của các cặp đảo gần nhau. Vì vậy, lãnh hải của một đảo sẽ là một hình đa giác. Cách phân chia dựa trên khoảng cách tối điểm gần nhất này trong toán học được gọi là Voronoi diagram². Trong không gian ba chiều, lấy ví dụ là các hành tinh, lãnh không của mỗi hành tinh sẽ là một đa diện. Trong không gian nhiều chiều hơn, chúng ta sẽ có những siêu đa diện (*hyperpolygon*).

Quay lại với bài toán phân nhóm và cụ thể là thuật toán K -means clustering, chúng ta cùng thảo luận cơ sở toán học, cách xây dựng và tối ưu hàm mất mát của nó.

10.2 Phân tích toán học

Mục đích cuối cùng của thuật toán K -means clustering là từ dữ liệu đầu vào và số lượng nhóm cần tìm, hãy xác định centroid của mỗi nhóm và phân các điểm dữ liệu vào các nhóm tương ứng. Giả sử thêm rằng mỗi điểm dữ liệu chỉ thuộc vào đúng một nhóm.

Giả sử N điểm dữ liệu trong training set được ghép lại thành $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ và $K < N$ là số cluster được xác định trước. Ta cần tìm các centroid $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_K \in \mathbb{R}^{d \times 1}$ và label của mỗi điểm dữ liệu. Ở đây, mỗi cluster được đại diện bởi một label, thường là một số tự nhiên từ 1 đến K . Nhắc lại rằng các điểm dữ liệu trong bài toán K -means clustering ban đầu không có label cụ thể, nhiệm vụ của ta là đi tìm label của chúng sao cho các điểm có cùng label nằm gần nhau, tạo thành một cluster.

² Voronoi diagram–Wikipedia (<https://goo.gl/xReCW8>).

Với mỗi điểm dữ liệu \mathbf{x}_i , ta cần tìm label $y_i = k$ của nó, ở đây $k \in \{1, 2, \dots, K\}$. Một kỹ thuật khác khác thường được dùng để biểu diễn label này có tên là *one-hot coding*. Mỗi label k được thay thế bằng một *vector hàng* $\mathbf{y}_i \in \mathbb{R}^{1 \times K}$ —được gọi là *label vector*, trong đó tất cả các phần tử của \mathbf{y}_i bằng 0, ngoại trừ phần tử ở vị trí thứ k bằng 1. Cụ thể, $y_{ij} = 0, \forall j \neq k, y_{ik} = 1$. Khi *chồng* các vector \mathbf{y}_i lên nhau, ta được một ma trận label $\mathbf{Y} \in \mathbb{R}^{N \times K}$. Nhắc lại rằng y_{ij} là phần tử hàng thứ i , cột thứ j của ma trận \mathbf{Y} , và nó cũng chính là phần tử thứ j của vector \mathbf{y}_i . Ví dụ, nếu một điểm dữ liệu có label vector là $[1, 0, 0, \dots, 0]$ thì nó thuộc vào cluster thứ nhất, là $[0, 1, 0, \dots, 0]$ thì nó thuộc vào cluster thứ hai, v.v. Ràng buộc của \mathbf{y}_i có thể viết dưới dạng toán học như sau:

$$y_{ij} \in \{0, 1\}, \forall i, j; \quad \sum_{j=1}^K y_{ij} = 1, \forall i \quad (10.1)$$

10.2.1 Hàm mất mát và bài toán tối ưu

Nếu gọi $\mathbf{m}_k \in \mathbb{R}^d$ là centroid của mỗi cluster và thay thế tất cả các điểm được phân vào cluster này bởi \mathbf{m}_k , một điểm dữ liệu \mathbf{x}_i được phân vào cluster k sẽ bị sai số là $(\mathbf{x}_i - \mathbf{m}_k)$. Chúng ta mong muốn vector sai số này gần với vector không, tức \mathbf{x}_i gần với \mathbf{m}_k . Một đại lượng đơn giản giúp đo khoảng cách giữa hai điểm là (bình phương) khoảng cách Euclid $\|\mathbf{x}_i - \mathbf{m}_k\|_2^2$. Hơn nữa, vì \mathbf{x}_i được phân vào cluster k nên $y_{ik} = 1, y_{ij} = 0, \forall j \neq k$. Khi đó, biểu thức khoảng cách Euclid có thể được viết lại thành

$$\|\mathbf{x}_i - \mathbf{m}_k\|_2^2 = y_{ik} \|\mathbf{x}_i - \mathbf{m}_k\|_2^2 = \sum_{j=1}^K y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (10.2)$$

Như vậy, sai số trung bình cho toàn bộ dữ liệu sẽ là:

$$\mathcal{L}(\mathbf{Y}, \mathbf{M}) = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^K y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (10.3)$$

Trong đó $\mathbf{M} = [\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_K] \in \mathbb{R}^{d \times K}$ là ma trận tạo bởi K centroid. Hàm mất mát trong bài toán K -means clustering là $\mathcal{L}(\mathbf{Y}, \mathbf{M})$ với ràng buộc như được nêu trong (10.1). Tóm lại, bài toán cần tối ưu là

$$\mathbf{Y}, \mathbf{M} = \underset{\mathbf{Y}, \mathbf{M}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^K y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (10.4)$$

thoả mãn: $y_{ij} \in \{0, 1\}, \forall i, j; \quad \sum_{j=1}^K y_{ij} = 1, \forall i$

10.2.2 Thuật toán tối ưu hàm mất mát

Bài toán (10.4) là một bài toán khó tìm *điểm tối ưu* vì nó có thêm các điều kiện ràng buộc. *Bài toán này thuộc loại mix-integer programming (điều kiện biến là số nguyên) - là loại rất*

khó tìm nghiệm tối ưu toàn cục. Tuy nhiên, trong một số trường hợp chúng ta vẫn có thể tìm được phương pháp để tìm được nghiệm gần đúng. Một kỹ thuật đơn giản và phổ biến để giải bài toán (10.4) là xen kẽ giải \mathbf{Y} và \mathbf{M} khi biến còn lại được cố định tới khi hàm mất mát hội tụ. Chúng ta sẽ lần lượt giải quyết hai bài toán sau.

Cố định \mathbf{M} , tìm \mathbf{Y}

Giả sử đã tìm được các centroid, hãy tìm các label vector để hàm mất mát đạt giá trị nhỏ nhất. Điều này tương đương với việc tìm cluster cho mỗi điểm dữ liệu. Khi các centroid là cố định, bài toán tìm label vector cho toàn bộ dữ liệu có thể được chia nhỏ thành bài toán tìm label vector cho từng điểm dữ liệu \mathbf{x}_i như sau:

$$\mathbf{y}_i = \underset{\mathbf{y}_i}{\operatorname{argmin}} \frac{1}{N} \sum_{j=1}^K y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (10.5)$$

$$\text{thoả mãn: } y_{ij} \in \{0, 1\}, \forall i, j; \quad \sum_{j=1}^K y_{ij} = 1, \forall i$$

Vì chỉ có một phần tử của label vector \mathbf{y}_i bằng 1 nên bài toán (10.5) chính là bài toán đi tìm centroid gần điểm \mathbf{x}_i nhất: $j = \operatorname{argmin}_j \|\mathbf{x}_i - \mathbf{m}_j\|_2^2$.

Vì $\|\mathbf{x}_i - \mathbf{m}_j\|_2^2$ chính là bình phương khoảng cách Euclid từ điểm \mathbf{x}_i tới centroid \mathbf{m}_j , ta có thể kết luận rằng **mỗi điểm \mathbf{x}_i thuộc vào cluster có centroid gần nó nhất!** Từ đó ta có thể suy ra label vector của từng điểm dữ liệu.

Cố định \mathbf{Y} , tìm \mathbf{M}

Giả sử đã tìm được cluster cho từng điểm, hãy tìm centroid mới cho mỗi cluster để hàm mất mát đạt giá trị nhỏ nhất.

Một khi label vector cho từng điểm dữ liệu đã được xác định, bài toán tìm centroid cho mỗi cluster được rút gọn thành

$$\mathbf{m}_j = \underset{\mathbf{m}_j}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^N y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2. \quad (10.6)$$

Tới đây, ta có thể tìm nghiệm bằng phương pháp giải phương trình đạo hàm bằng không, vì hàm cần tối ưu là một hàm liên tục và có đạo hàm xác định tại mọi điểm \mathbf{m}_j . Đặt $l(\mathbf{m}_j)$ là hàm bên trong dấu argmin trong 10.6, ta cần giải phương trình

$$\nabla_{\mathbf{m}_j} l(\mathbf{m}_j) = \frac{2}{N} \sum_{i=1}^N y_{ij} (\mathbf{m}_j - \mathbf{x}_i) = \mathbf{0} \Leftrightarrow \mathbf{m}_j \sum_{i=1}^N y_{ij} = \sum_{i=1}^N y_{ij} \mathbf{x}_i \Leftrightarrow \mathbf{m}_j = \frac{\sum_{i=1}^N y_{ij} \mathbf{x}_i}{\sum_{i=1}^N y_{ij}} \quad (10.7)$$

Nếu để ý một chút, chúng ta sẽ thấy rằng mẫu số chính là phép đếm số lượng các điểm dữ liệu trong cluster j . Còn tử số chính là tổng các điểm dữ liệu trong cluster j . Nói cách khác, \mathbf{m}_j là trung bình cộng (mean) của các điểm trong cluster j .

Tên gọi *K-means clustering* cũng xuất phát từ đây.

10.2.3 Tóm tắt thuật toán

Tới đây, ta có thể tóm tắt thuật toán K-means clustering như sau.

Thuật toán 10.1: K-means clustering

Đầu vào: Ma trận dữ liệu $\mathbf{X} \in \mathbb{R}^{d \times N}$ và số lượng cluster cần tìm $K < N$.

Đầu ra: Ma trận các centroid $\mathbf{M} \in \mathbb{R}^{d \times K}$ và ma trận label $\mathbf{Y} \in \mathbb{R}^{N \times K}$.

1. Chọn K điểm bất kỳ trong training set làm các centroid ban đầu.
2. Phân mỗi điểm dữ liệu vào cluster có centroid gần nó nhất.
3. Nếu việc phân nhóm dữ liệu vào từng cluster ở bước 2 không thay đổi so với vòng lặp trước nó thì ta dừng thuật toán.
4. Cập nhật centroid cho từng cluster bằng cách lấy trung bình cộng của tất cả các điểm dữ liệu đã được gán vào cluster đó sau bước 2.
5. Quay lại bước 2.

Thuật toán này được đảm bảo sẽ hội tụ sau một số hữu hạn vòng lặp. Thật vậy, vì hàm mất mát là một số dương và sau mỗi bước 2 hoặc 3, giá trị của hàm mất mát bị giảm đi. Vậy, dãy số biểu diễn giá trị của hàm mất mát sau mỗi bước là một đại lượng không tăng và bị chặn dưới, điều này chỉ ra rằng dãy số này phải hội tụ. Để ý thêm nữa, số lượng cách phân nhóm cho toàn bộ dữ liệu là hữu hạn (khi số cluster K là cố định) nên đến một lúc nào đó, hàm mất mát sẽ không thể thay đổi, và chúng ta có thể dừng thuật toán tại đây.

Nếu tồn tại một cluster không chứa điểm nào, mẫu số trong (10.7) sẽ bằng không, và phép chia sẽ không thực hiện được. Vì vậy, K điểm bất kỳ trong training set được chọn làm các centroid ban đầu ở Bước 1. để đảm bảo rằng mỗi cluster có ít nhất một điểm. Trong quá trình huấn luyện, nếu tồn tại một cluster không chứa điểm nào, có hai cách giải quyết. Cách thứ nhất là bỏ đi cluster đó và giảm K đi một. Cách thứ hai là thay centroid của cluster đó bằng một điểm bất kỳ trong training set, chẳng hạn, điểm xa centroid hiện tại của nó nhất.

10.3 Ví dụ trên Python

10.3.1 Giới thiệu bài toán

Chúng ta sẽ làm một ví dụ đơn giản. Trước hết, ta tạo centroid và dữ liệu cho từng cluster bằng cách lấy mẫu theo phân phối chuẩn có kỳ vọng là centroid của cluster đó và ma trận hiệp phương sai là ma trận đơn vị. Ở đây, hàm `cdist` trong `scipy.spatial.distance` được dùng để tính khoảng cách giữa các cặp điểm trong hai tập hợp một cách hiệu quả³.

Dữ liệu được tạo bằng cách lấy ngẫu nhiên 500 điểm cho mỗi cluster theo phân phối chuẩn có kỳ vọng lần lượt là (2, 2), (8, 3) và (3, 6), ma trận hiệp phương sai giống nhau và là ma trận đơn vị.

³ việc xây dựng hàm số này không sử dụng thư viện đã được thảo luận kỹ trong Chương 9

```

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
import random
np.random.seed(18)

means = [[2, 2], [8, 3], [3, 6]]
cov = [[1, 0], [0, 1]]
N = 500
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X2 = np.random.multivariate_normal(means[2], cov, N)

X = np.concatenate((X0, X1, X2), axis = 0)
K = 3 # 3 clusters
original_label = np.asarray([0]*N + [1]*N + [2]*N).T

```

10.3.2 Các hàm số cần thiết cho K -means clustering

Trước khi viết thuật toán chính K -means clustering, ta cần viết một số hàm phụ trợ:

1. `kmeans_init_centroids` để khởi tạo các centroids ban đầu.
2. `kmeans_assign_labels` để tìm label mới cho các điểm khi cố định các centroid.
3. `kmeans_update_centroids` để cập nhật các centroid khi biết label của mỗi điểm dữ liệu.
4. `has_converged` để kiểm tra điều kiện dừng của thuật toán.

```

def kmeans_init_centroids(X, k):
    # randomly pick k rows of X as initial centroids
    return X[np.random.choice(X.shape[0], k, replace=False)]

def kmeans_assign_labels(X, centroids):
    # calculate pairwise distances btw data and centroids
    D = cdist(X, centroids)
    # return index of the closest centroid
    return np.argmin(D, axis = 1)

def has_converged(centroids, new_centroids):
    # return True if two sets of centroids are the same
    return (set([tuple(a) for a in centroids]) ==
            set([tuple(a) for a in new_centroids]))

def kmeans_update_centroids(X, labels, K):
    centroids = np.zeros((K, X.shape[1]))
    for k in range(K):
        # collect all points that are assigned to the k-th cluster
        Xk = X[labels == k, :]
        centroids[k, :] = np.mean(Xk, axis = 0) # then take average
    return centroids

```

Phần chính của K -means clustering:

```
def kmeans(X, K):
    centroids = [kmeans_init_centroids(X, K)]
    labels = []
    it = 0
    while True:
        labels.append(kmeans_assign_labels(X, centroids[-1]))
        new_centroids = kmeans_update_centroids(X, labels[-1], K)
        if has_converged(centroids[-1], new_centroids):
            break
        centroids.append(new_centroids)
        it += 1
    return (centroids, labels, it)
```

Áp dụng thuật toán vừa viết vào dữ liệu ban đầu, hiển thị kết quả cuối cùng.

```
(centroids, labels, it) = kmeans(X, K)
print('Centers found by our algorithm:\n', centroids[-1])
kmeans_display(X, labels[-1])
```

Kết quả:

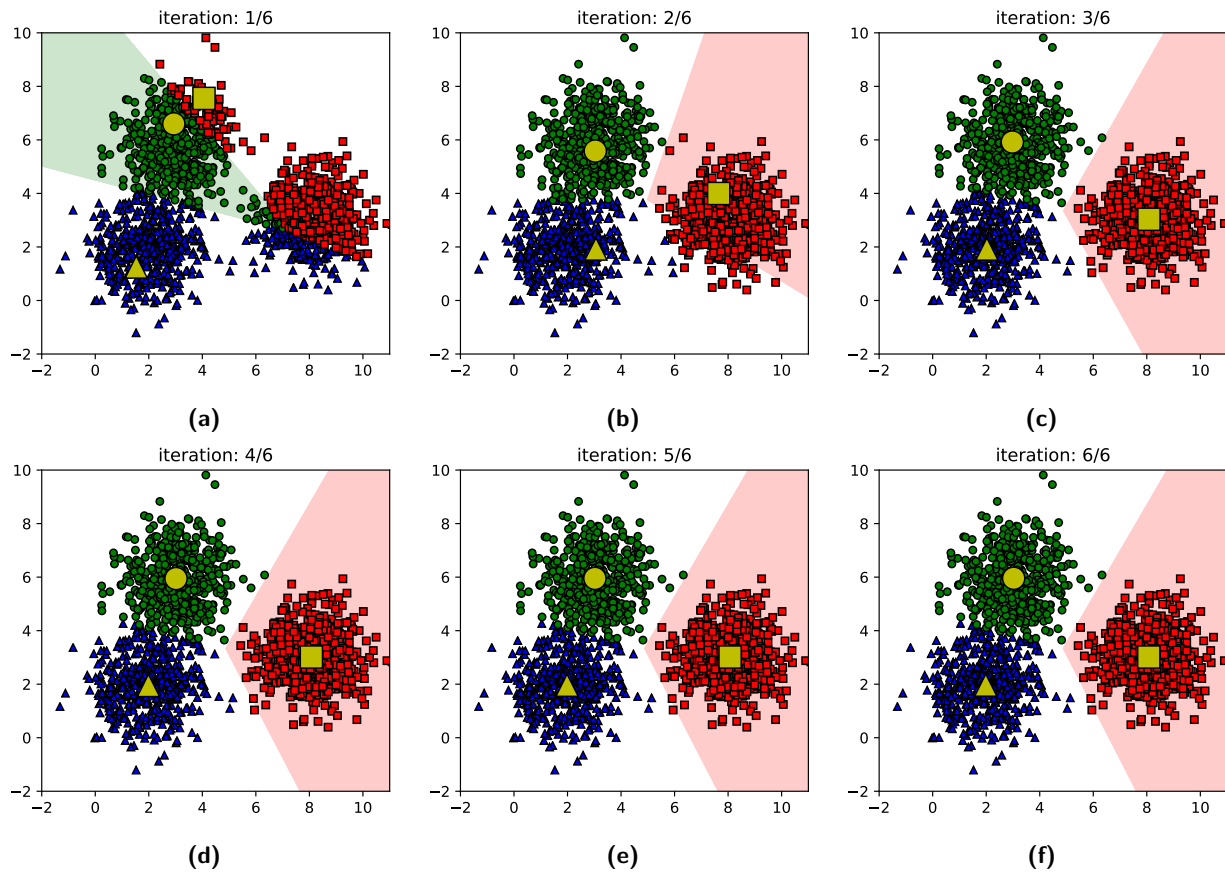
```
Centers found by our algorithm:
[[ 1.9834967  1.96588127]
 [ 3.02702878  5.95686115]
 [ 8.07476866  3.01494931]]
```

Hình 10.2 minh họa thuật toán K -means clustering trên tập dữ liệu này sau một số vòng lặp. Ta nhận thấy rằng centroid và các vùng *lãnh thổ* của chúng thay đổi qua các vòng lặp và hội tụ sau chỉ sáu vòng lặp. Từ kết quả này chúng ta thấy rằng thuật toán K -means clustering làm việc khá thành công, các centroid tìm được gần với các centroid ban đầu, và các nhóm dữ liệu được phân ra gần như hoàn hảo (một vài điểm gần ranh giới giữa hai cluster xanh có thể đã lẫn vào nhau).

10.3.3 Kết quả tìm được bằng thư viện scikit-learn

Để kiểm tra thêm, chúng ta hãy so sánh kết quả trên với kết quả thu được bằng cách sử dụng thư viện **scikit-learn**.

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters=3, random_state=0).fit(X)
print('Centers found by scikit-learn:')
print(model.cluster_centers_)
pred_label = model.predict(X)
kmeans_display(X, pred_label)
```



Hình 10.2: Thuật toán K -means clustering qua các vòng lặp.

Kết quả:

```
Centroids found by scikit-learn:
[[ 8.0410628  3.02094748]
 [ 2.99357611  6.03605255]
 [ 1.97634981  2.01123694]]
```

Ta nhận thấy rằng các centroid tìm được cũng rất gần với kết quả kỳ vọng. Từ các centroid này, cluster của mỗi điểm dữ liệu cũng dễ dàng được suy ra.

Tiếp theo, chúng ta cùng xem xét ba ứng dụng đơn giản của K -means clustering.

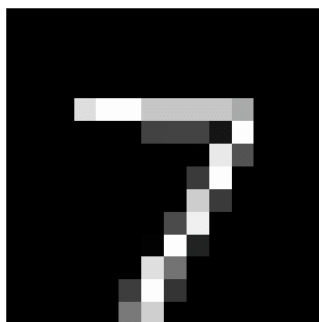
10.4 Phân nhóm chữ số viết tay

10.4.1 Bộ cơ sở dữ liệu MNIST

MNIST [LCB10] là bộ cơ sở dữ liệu lớn nhất về chữ số viết tay và được sử dụng trong hầu hết các thuật toán phân lớp hình ảnh. MNIST bao gồm hai tập con: training set có tổng



Hình 10.3: 200 mẫu ngẫu nhiên trong bộ cơ sở dữ liệu MNIST.



=

[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	222	254	254	198	198	198	198	170	0	0	0	0]
[0	0	0	0	0	0	66	67	67	21	254	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	233	83	0	0	0]
[0	0	0	0	0	0	0	0	0	59	254	0	0	0	0]
[0	0	0	0	0	0	0	0	205	58	0	0	0	0	0]
[0	0	0	0	0	0	0	75	240	0	0	0	0	0	0]
[0	0	0	0	0	0	0	3	254	35	0	0	0	0	0]
[0	0	0	0	0	0	224	115	0	0	0	0	0	0	0]
[0	0	0	0	0	61	254	52	0	0	0	0	0	0	0]
[0	0	0	0	0	121	207	0	0	0	0	0	0	0	0]

Hình 10.4: Ví dụ về chữ số 7 và giá trị các pixel của nó.

cộng 60 nghìn mẫu khác nhau về chữ số viết tay từ 0 đến 9, test set có 10 nghìn mẫu khác nhau. Tất cả đều đã được gán nhãn. Hình 10.3 hiển thị 200 mẫu được trích ra từ MNIST.

Mỗi bức ảnh là một ảnh xám (chỉ có một channel), có kích thước 28×28 pixel (tổng cộng 784 pixel). Mỗi pixel mang một giá trị là một số tự nhiên từ 0 đến 255. Các pixel màu đen có giá trị bằng không, các pixel càng trắng thì có giá trị càng cao, nhưng không quá 255. Hình 10.4 là một ví dụ về chữ số 7 và giá trị các pixel của nó. Vì mục đích hiển thị ma trận pixel ở bên phải, bức ảnh kích thước 28×28 ban đầu đã được resize về kích thước 14×14 .

10.4.2 Bài toán phân nhóm giả định

Bài toán: Giả sử rằng ta không biết nhãn của các chữ số này, hãy phân các bức ảnh gần giống nhau về một nhóm.

Bài toán này có thể được giải quyết bằng K-means clustering.

Mỗi bức ảnh được coi là một điểm dữ liệu. Vector đặc trưng của một bức ảnh đơn giản là vector cột cỡ 784×1 thu được bằng cách chồng các cột của bức ảnh ban đầu lên nhau.

10.4.3 Làm việc trên Python

Để tải về MNIST, chúng ta có thể dùng trực tiếp một hàm số trong scikit-learn như sau.

```

from __future__ import print_function
import numpy as np
from sklearn.datasets import fetch_mldata

data_dir = '../..data' # path to your data folder
mnist = fetch_mldata('MNIST original', data_home=data_dir)
print("Shape of minst data:", mnist.data.shape)

```

Kết quả:

```
Shape of minst data: (70000, 784)
```

shape của ma trận dữ liệu `mnist.data` là **(70000, 784)** tức có 70000 mẫu, mỗi mẫu có kích thước 784. Chú ý rằng trong scikit-learn, mỗi điểm dữ liệu thường được lưu dưới dạng một vector hàng. Tiếp theo, chúng ta lấy ra ngẫu nhiên 10000 mẫu và thực hiện K-means cluster trên tập con này.

```

from sklearn.cluster import KMeans
from sklearn.neighbors import NearestNeighbors
K = 10 # number of clusters
N = 10000
X = mnist.data[np.random.choice(mnist.data.shape[0], N)]
kmeans = KMeans(n_clusters=K).fit(X)
pred_label = kmeans.predict(X)

```

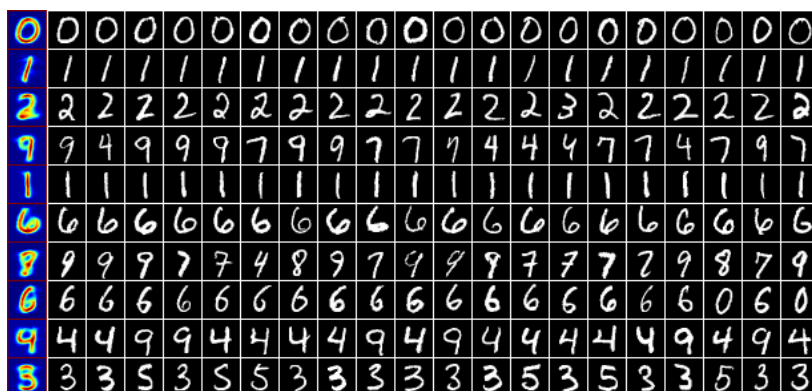
Sau khi thực hiện đoạn code trên, các centroid được lưu trong biến `kmeans.cluster_centers_`, label của mỗi điểm dữ liệu được lưu trong biến `pred_label`. Hình 10.5 hiển thị các centroid tìm được và 20 mẫu ngẫu nhiên được phân vào mỗi cluster tương ứng. Mỗi hàng tương ứng với một cluster, cột đầu tiên có nền xanh bên trái là các centroid tìm được (màu đỏ hơn là các pixel có giá trị cao hơn). Chúng ta thấy rằng các centroid đều hoặc là giống với một chữ số nào đó, hoặc là kết hợp của hai/ba chữ số nào đó. Ví dụ, centroid ở hàng thứ 4 là sự kết hợp của các số 4, 7, 9; ở hàng thứ 7 là kết hợp của chữ số 7, 8 và 9.

Quan sát thấy các bức ảnh lấy ra ngẫu nhiên từ mỗi cluster trông không thực sự giống nhau. Lý do có thể là những bức ảnh này ở xa các centroid mặc dù centroid đó đã là gần nhất. Như vậy K-means clustering làm việc chưa thực sự tốt trong trường hợp này. Tuy nhiên, chúng ta vẫn có thể khai thác một số thông tin hữu ích sau khi thực hiện thuật toán này. Thay vì chọn ngẫu nhiên các bức ảnh trong mỗi cluster, ta chọn 20 bức ảnh gần centroid của mỗi cluster nhất, vì càng gần centroid thì độ tin cậy càng cao. Hãy quan sát Hình 10.6. Ta có thể thấy dữ liệu trong mỗi hàng khá giống nhau và giống với centroid ở cột đầu tiên bên trái. Có một vài quan sát thú vị có thể rút ra từ đây:

1. Có hai kiểu viết chữ số 1—thẳng và chéo. Và K-means clustering nghĩ rằng đó là hai chữ số khác nhau. Điều này là dễ hiểu vì K-means clustering là một thuật toán unsupervised learning. Nếu có sự can thiệp của con người, chúng có thể được nhóm lại thành một.



Hình 10.5: Hiển thị các centroid (cột đầu) và 20 điểm ngẫu nhiên được phân vào từng cluster. Các chữ số trên mỗi hàng thuộc vào cùng một cluster.



Hình 10.6: Hiển thị các centroid (cột đầu) và 20 điểm gần centroid nhất được phân vào từng cluster. Các chữ số trên mỗi hàng thuộc vào cùng một cluster.

- Ở hàng thứ chín, chữ số 4 và 9 được phân vào cùng một cluster. Sự thật là hai chữ số này khá giống nhau. Điều tương tự xảy ra đối với hàng thứ bảy với các chữ số 7, 8, 9. K-means clustering có thể được áp dụng để tiếp tục phân nhỏ các cluster đó.

Trong clustering có một kỹ thuật thường được sử dụng là *clustering phân tầng* (*hierarchical clustering* [Ble08]). Có hai loại hierachical clustering:

- **Agglomerative** tức “đi từ dưới lên”. Ban đầu coi một vài điểm dữ liệu gần nhau là một cluster, sau đó các cặp cluster gần giống nhau được gộp lại làm một cluster lớn hơn. Ở đây, sự giống nhau của hai cluster có thể được xác định dựa trên khoảng cách giữa hai centroid tương ứng. Cụ thể hơn, ban đầu ta chọn K là một số lớn gần bằng số điểm dữ liệu. Sau khi thực hiện K -means clustering lần đầu, các cluster gần nhau được ghép lại thành một cluster. Sau bước này, ta được một số lượng cluster nhỏ hơn. Ta tiếp tục làm K -means clustering với điểm khởi tạo là centroid của các nhóm vừa thu được. Lặp lại quá trình này đến khi nhận được kết quả chấp nhận được.
- **Divisive** tức “đi từ trên xuống”. Ban đầu coi tất cả các điểm dữ liệu thuộc cùng một cluster, sau đó chia nhỏ mỗi cluster bằng một thuật toán clustering nào đó. Việc này có thể được thực hiện bằng cách ban đầu chọn một số K nhỏ làm số lượng cluster, sau đó trong mỗi cluster thu được, ta tiếp tục làm K -means clustering. Tiếp tục quá trình cho tới khi được kết quả chấp nhận được.



Hình 10.7: Ảnh:Trọng Vũ (<https://goo.gl/9D8aXW>) .

10.5 Tách vật thể trong ảnh

K-means clustering có thể được áp dụng vào một bài toán xử lý ảnh khác, bài toán *tách vật thể trong ảnh* (*object segmentation*). Cho bức ảnh như trong Hình 10.7, hãy xây dựng một thuật toán tự động nhận ra vùng khuôn mặt và tách nó ra.

Bức ảnh có ba màu chủ đạo: hồng ở khăn và môi; đen ở mắt, tóc, và hậu cảnh; màu da ở vùng còn lại của khuôn mặt. Ảnh này khá rõ nét và các vùng được phân biệt rõ ràng bởi màu sắc nên chúng ta có thể áp dụng thuật toán K-means clustering. Thuật toán này sẽ phân các pixel ảnh thành ba cluster, cluster chứa phần khuôn mặt có thể được chọn, có thể bằng tay.

Đây là một bức ảnh màu, mỗi điểm ảnh sẽ được biểu diễn bởi ba giá trị tương ứng với màu Red, Green, và Blue, mỗi giá trị này cũng là một số tự nhiên không vượt quá 255. Nếu coi mỗi pixel là một điểm dữ liệu mô tả bởi một vector ba chiều chứa các giá trị này, sau đó áp dụng thuật toán K-means clustering, chúng ta có thể có kết quả mong muốn.

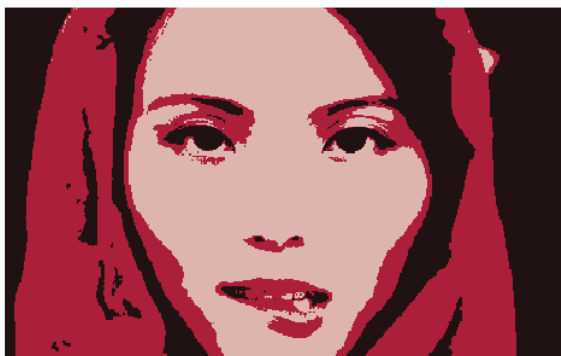
10.5.1 Làm việc trên Python

Khai báo thư viện và load bức ảnh:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans
img = mpimg.imread('girl3.jpg')
plt.imshow(img)
imgplot = plt.imshow(img)
plt.axis('off')
plt.show()
```

Biến đổi bức ảnh thành 1 ma trận mà mỗi hàng là 1 pixel với 3 giá trị màu

```
X = img.reshape((img.shape[0]*img.shape[1], img.shape[2]))
```



Hình 10.8: Kết quả nhận được sau khi thực hiện K-means clustering cho các điểm dữ liệu. Có ba cluster tương ứng với ba màu đỏ, hồng, đen.

Phần còn lại của source code có thể được tìm thấy tại <https://goo.gl/Tn6Gec>.

Sau khi tìm được các cluster, giá trị của mỗi pixel được thay bằng giá trị của centroid tương ứng. Kết quả được minh họa trên Hình 10.8. Ba màu đỏ, đen, và màu da đã được phân nhóm khá thành công. Và khuôn mặt có thể được tách ra từ phần có màu da (và vùng bên trong nó). Như vậy, K-means clustering tạo ra một kết quả chấp nhận được cho bài toán này.

10.6 Image Compression (nén ảnh và nén dữ liệu nói chung)

Trước hết, xét đoạn code dưới đây.

```
for K in [5, 10, 15, 20]:
    kmeans = KMeans(n_clusters=K).fit(X)
    label = kmeans.predict(X)

    img4 = np.zeros_like(X)
    # replace each pixel by its centroid
    for k in range(K):
        img4[label == k] = kmeans.cluster_centroids_[k]
    # reshape and display output image
    img5 = img4.reshape((img.shape[0], img.shape[1], img.shape[2]))
    plt.imshow(img5, interpolation='nearest')
    plt.axis('off')
    plt.show()
```

Giải thích: Để ý thấy rằng mỗi một pixel có thể nhận một trong số $256^3 = 16,777,216$ (16 triệu màu). Đây là một số rất lớn (tương đương với 24 bit cho một điểm ảnh). Nếu ta muốn lưu mỗi điểm ảnh với một số bit nhỏ hơn và chấp nhận mất dữ liệu ở một mức nào đó, K-means clustering là một giải pháp đơn giản cho việc này. Trong bài toán segmentation phía trên, có ba cluster, và mỗi một điểm ảnh sau khi xử lý sẽ được biểu diễn bởi một số tương ứng với một cluster. Tuy nhiên, chất lượng bức ảnh rõ ràng đã giảm đi nhiều. Trong đoạn code trên đây, ta đã làm một thí nghiệm nhỏ với số lượng cluster được tăng lên 5, 10, 15, 20. Sau khi tìm được centroid cho mỗi cluster, giá trị của một điểm ảnh được thay bằng giá trị của centroid tương ứng. Kết quả được cho trên Hình 10.9. Ta có thể quan sát thấy



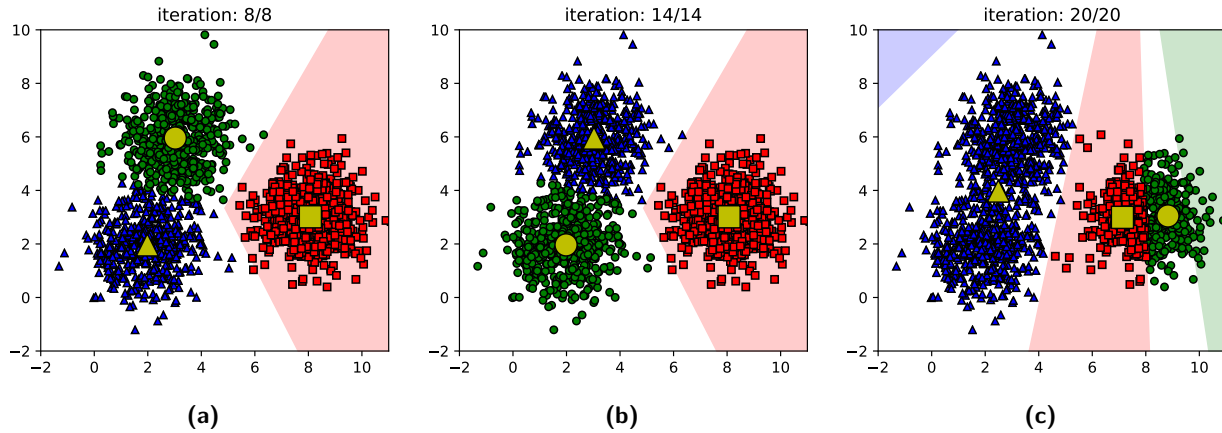
Hình 10.9: Chất lượng nén ảnh với số lượng cluster khác nhau.

ràng khi số lượng cluster tăng lên, chất lượng bức ảnh đã được cải thiện. Để nén bức ảnh này, ta chỉ cần lưu K centroid tìm được và label của mỗi điểm ảnh.

10.7 Thảo luận

10.7.1 Hạn chế của K -means clustering

- **Số lượng cluster K cần được xác định trước.** Trong thực tế, nhiều trường hợp chúng ta không xác định được giá trị này. Bạn đọc có thể tham khảo một cách giúp xác định giá trị K này có tên là elbow method (<https://goo.gl/euYhpK>).
- **Nghiệm cuối cùng phụ thuộc vào các centroid được khởi tạo ban đầu.** Trong thuật toán này, hàm khởi tạo `kmeans_init_centroids` chọn ngẫu nhiên K điểm từ tập dữ liệu làm các centroid ban đầu. Thêm nữa, thuật toán K -means clustering không đảm bảo tìm được nghiệm tối ưu toàn cục, nên nghiệm cuối cùng phụ thuộc rất nhiều vào các centroid được khởi tạo ban đầu. Hình 10.10 thể hiện các kết quả khác nhau khi các centroid được khởi tạo khác nhau. Ta cũng thấy rằng trường hợp (a) và (b) cho kết quả tốt, trong khi kết quả thu được ở trường hợp (c) không thực sự tốt. Một điểm nữa có thể



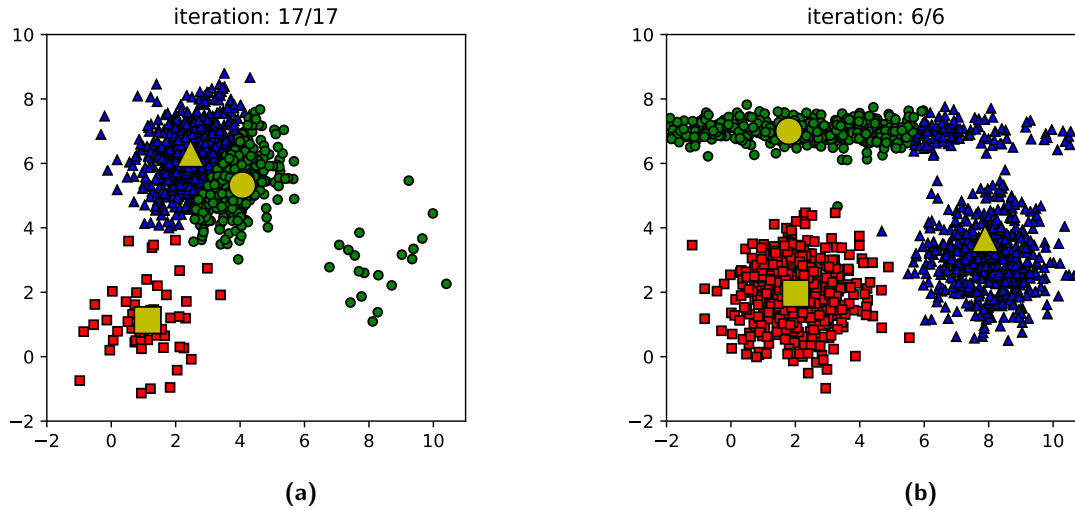
Hình 10.10: Các giá trị khởi tạo ban đầu khác nhau dẫn đến các nghiệm khác nhau.

rút ra là số lượng vòng lặp tới khi thuật toán hội tụ cũng khác nhau. Trường hợp (a) và (b) cùng cho kết quả tốt nhưng (b) chạy trong thời gian gần gấp đôi. Một kỹ thuật giúp hạn chế nghiệm xấu như trường hợp (c) là chạy thuật toán K -means clustering nhiều lần với các centroid được khởi tạo khác nhau và chọn ra lần chạy cho giá trị hàm mất mát thấp nhất⁴. Ngoài ra, [KA04], Kmeans++ [AV07, BMV⁺12] cũng là một vài thuật toán nổi tiếng giúp chọn các centroid ban đầu.

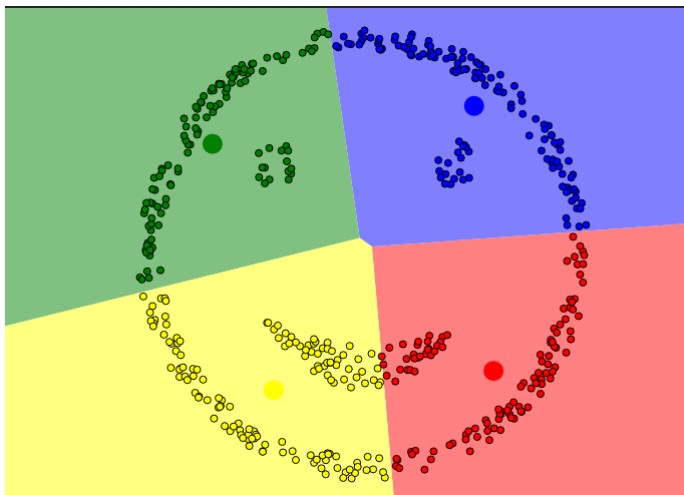
- **Các cluster cần có số lượng điểm gần bằng nhau.** Hình 10.11a minh họa kết quả khi các cluster có số lượng điểm chênh lệch. Trong trường hợp này, nhiều điểm lẻ ra thuộc cluster xanh lam đã bị phân nhầm vào cluster xanh lục.
- **Các cluster cần có dạng hình tròn (cầu)** Khi các cluster vẫn tuân theo phân phối chuẩn nhưng ma trận hiệp phương sai không tỉ lệ với ma trận đơn vị, các cluster sẽ có dạng không phải là tròn (hoặc cầu trong không gian nhiều chiều). Khi đó, K -means clustering cũng không hoạt động hiệu quả. Lý do chính là vì K -means clustering quyết định cluster của một điểm dữ liệu dựa trên khoảng cách Euclid của nó tới các centroid. Trong trường hợp này, Gaussian mixture models (GMM) [Rey15] có thể cho kết quả tốt hơn⁵. Trong GMM, mỗi cluster được giả sử tuân theo một phân phối chuẩn với ma trận hiệp phương sai không nhất thiết tỉ lệ với ma trận đơn vị. Ngoài các centroid, các ma trận hiệp phương sai cũng là các biến cần tối ưu trong GMM.
- **Khi một cluster bị bao bọc bởi một cluster khác** Hình 10.12 là một ví dụ kinh điển về việc K -means clustering không thể phân cụm dữ liệu. Một cách tự nhiên, chúng ta sẽ phân dữ liệu ra thành bốn cluster: mắt trái, mắt phải, miệng, xung quanh mặt. Nhưng vì mắt và miệng nằm trong khuôn mặt nên K -means clustering cho kết quả không chính xác. Với dữ liệu như trong ví dụ này, spectral clustering [VL07, NJW02] sẽ cho kết quả tốt hơn. Spectral clustering cũng coi các điểm gần nhau tạo thành một cluster, nhưng không giả sử về một centroid chung cho cả cluster. Spectral clustering được thực hiện dựa trên

⁴ $KMeans$ -scikit-learn (<https://goo.gl/5KavVn>).

⁵ Đọc thêm: *Gaussian mixture models*-Wikipedia (<https://goo.gl/GzdauR>).



Hình 10.11: K -means clustering hoạt động không thực sự tốt trong trường hợp các cluster có số lượng phần tử chênh lệch hoặc các cluster không có dạng hình tròn (cầu).



Hình 10.12: Một ví dụ về việc K -means clustering phân nhóm sai.

một đồ thị vô hướng với đỉnh là các điểm dữ liệu và cạnh được nối giữa các điểm gần nhau, mỗi cạnh được đánh trọng số là một hàm của khoảng cách giữa hai điểm.

10.7.2 Các ứng dụng khác của K -means clustering

Mặc dù có những hạn chế, K -means clustering vẫn cực kỳ quan trọng trong machine learning và là nền tảng cho nhiều thuật toán phức tạp khác. Dưới đây là một vài ứng dụng khác của K -means clustering.

1. Cách thay một điểm dữ liệu bằng centroid tương ứng là một trong số các kỹ thuật có tên chung là *Vector Quantization – VQ* [AM93]). Không chỉ trong nén dữ liệu, VQ còn được kết hợp với Bag-of-Words [LSP06] áp dụng rộng rãi trong các thuật toán xây dựng vector đặc trưng cho các bài toán phân loại.

2. Ngoài ra, VQ còn được áp dụng trong các bài toán tìm kiếm trong cơ sở dữ liệu lớn. Khi lượng điểm dữ liệu rất lớn, việc tìm kiếm trở nên cực kỳ quan trọng. Khó khăn chính của việc này là làm thế nào có thể tìm kiếm một cách nhanh chóng trong lượng dữ liệu khổng lồ đó. Ý tưởng cơ bản là sử dụng các thuật toán clustering để phân các điểm dữ liệu thành nhiều nhóm nhỏ và xấp xỉ mỗi điểm dữ liệu bằng centroid tương ứng. Khi tìm điểm gần nhất của một điểm *truy vấn* (*query*), thay vì tính khoảng cách giữa điểm truy vấn đó đến từng điểm trong cơ sở dữ liệu, ta sẽ chỉ cần tính khoảng cách từ điểm đó tới các centroid (số lượng nhỏ hơn). Sau đó trả về các điểm được phân vào centroid đó. Bạn đọc có thể đọc thêm các bài báo nổi tiếng gần đây về vấn đề này: Product Quantization [JDS11], Cartesian k-means [NF13, JDJ17], Composite Quantization [ZDW14], Additive Quantization [BL14].

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/QgW5f2>.

10.7.3 Đọc thêm

1. *Clustering documents using k-means–scikit-learn* (<https://goo.gl/y4xxy2>).
2. *Voronoi Diagram - Wikipedia* (<https://goo.gl/v8WQEv>).
3. *Cluster centroid initialization algorithm for K-means clustering* (<https://goo.gl/hBdody>).
4. *Visualizing K-Means Clustering* (<https://goo.gl/ULbpUM>).
5. *Visualizing K-Means Clustering - Stanford* (<https://goo.gl/idzR2i>).

Naive Bayes classifier

11.1 Naive Bayes classifier

Xét các bài toán phân lớp với C class khác nhau. Thay vì tìm ra chính xác label của mỗi điểm dữ liệu $\mathbf{x} \in \mathbb{R}^d$, ta có thể đi tìm xác suất để đầu ra đó rơi vào mỗi class: $p(y = c|\mathbf{x})$, hoặc viết gọn thành $p(c|\mathbf{x})$. Biểu thức này được hiểu là xác suất để đầu ra là class c biết rằng đầu vào là vector \mathbf{x} . Biểu thức này, nếu tính được, có thể giúp xác định class của mỗi điểm dữ liệu bằng cách chọn ra class có xác suất rơi vào cao nhất:

$$c = \operatorname{argmax}_{c \in \{1, \dots, C\}} p(c|\mathbf{x}) \quad (11.1)$$

Biểu thức trong dấu argmax ở (11.1) nhìn chung khó có cách tính trực tiếp. Thay vào đó, quy tắc Bayes thường được sử dụng:

$$c = \operatorname{argmax}_c p(c|\mathbf{x}) = \operatorname{argmax}_c \frac{p(\mathbf{x}|c)p(c)}{p(\mathbf{x})} = \operatorname{argmax}_c p(\mathbf{x}|c)p(c) \quad (11.2)$$

Dấu bằng thứ hai xảy ra theo quy tắc Bayes, dấu bằng thứ ba xảy ra vì $p(\mathbf{x})$ ở mẫu số không phụ thuộc vào c . Tiếp tục quan sát, $p(c)$ có thể được hiểu là xác suất để một điểm *bất kỳ* rơi vào class c . Nếu training set lớn, nó có thể được xác định bằng maximum likelihood estimation (MLE)—là tỉ lệ giữa số điểm thuộc class c và số điểm trong training set. Nếu training set nhỏ, giá trị này có thể được ước lượng bằng maximum a posteriori (MAP). Cách thứ nhất thường được sử dụng nhiều hơn.

Thành phần còn lại $p(\mathbf{x}|c)$, tức phân phối của các điểm dữ liệu trong class c , thường rất khó tính toán vì \mathbf{x} là một biến ngẫu nhiên nhiều chiều. Để có thể ước lượng được phân phối đó, training set phải rất lớn. Để giúp cho việc tính toán được đơn giản, người ta thường giả sử rằng các thành phần của biến ngẫu nhiên \mathbf{x} là độc lập với nhau khi đã biết c :

$$p(\mathbf{x}|c) = p(x_1, x_2, \dots, x_d|c) = \prod_{i=1}^d p(x_i|c) \quad (11.3)$$

Giả thiết các chiều của dữ liệu độc lập với nhau là quá chặt và trên thực tế, ít khi tìm được dữ liệu mà các thành phần hoàn toàn độc lập với nhau. Tuy nhiên, giả thiết *ngây thơ* (*naive*) này đôi khi mang lại những kết quả tốt bất ngờ. Giả thiết về sự độc lập của các chiều dữ liệu này được gọi là *naive Bayes*. Cách xác định label của dữ liệu dựa trên giả thiết này có tên là *naive Bayes classifier* (NBC).

NBC, nhờ vào tính đơn giản một cách *ngây thơ*, có tốc độ huấn luyện và kiểm thử rất nhanh. Việc này giúp nó mang lại hiệu quả cao trong các bài toán large-scale.

Ở bước huấn luyện, các phân phối $p(c)$ và $p(x_i|c), i = 1, \dots, d$ sẽ được xác định dựa vào dữ liệu huấn luyện. Việc xác định các giá trị này có thể dựa vào MLE hoặc MAP.

Ở bước kiểm thử, label của một điểm dữ liệu mới \mathbf{x} được xác định bởi

$$c = \operatorname{argmax}_{c \in \{1, \dots, C\}} p(c) \prod_{i=1}^d p(x_i|c) \quad (11.4)$$

Khi d lớn và các xác suất nhỏ, biểu thức ở vế phải của (11.4) là một số rất nhỏ, khi tính toán có thể gặp sai số. Để giải quyết việc này, (11.4) thường được viết lại dưới dạng tương đương bằng cách lấy log của vế phải:

$$c = \operatorname{argmax}_{c \in \{1, \dots, C\}} \left(\log(p(c)) + \sum_{i=1}^d \log(p(x_i|c)) \right) \quad (11.5)$$

Việc này không ảnh hưởng tới kết quả vì log là một hàm đồng biến trên tập các số dương.

Sự đơn giản của NBC mang lại hiệu quả đặc biệt trong các bài toán phân loại văn bản, ví dụ bài toán lọc tin nhắn hoặc email rác. Trong phần sau của chương này, chúng ta cùng xây dựng một bộ lọc email rác tiếng Anh đơn giản. Cả việc huấn luyện và kiểm thử của NBC là cực kỳ nhanh khi so với các phương pháp phân loại phức tạp khác. Việc giả sử các thành phần trong dữ liệu là độc lập với nhau khiến cho việc tính toán mỗi phân phối $p(\mathbf{x}_i|c)$ không mất nhiều thời gian.

Việc tính toán $p(\mathbf{x}_i|c)$ phụ thuộc vào loại dữ liệu. Có ba loại phân bố xác suất thường được sử dụng phổ biến là *Gaussian naive Bayes*, *multinomial naive Bayes*, và *Bernoulli Naive*. Chúng ta cùng xem xét vào từng loại.

11.2 Các phân phối thường dùng trong NBC

11.2.1 Gaussian naive Bayes

Mô hình này được sử dụng chủ yếu trong loại dữ liệu mà các thành phần là các biến liên tục. Với mỗi chiều dữ liệu i và một class c , x_i tuân theo một phân phối chuẩn có kỳ vọng μ_{ci} và phương sai σ_{ci}^2 :

$$p(x_i|c) = p(x_i|\mu_{ci}, \sigma_{ci}^2) = \frac{1}{\sqrt{2\pi\sigma_{ci}^2}} \exp\left(-\frac{(x_i - \mu_{ci})^2}{2\sigma_{ci}^2}\right) \quad (11.6)$$

Trong đó, bộ tham số $\theta = \{\mu_{ci}, \sigma_{ci}^2\}$ được xác định bằng MLE dựa trên các điểm trong training set thuộc class c .

11.2.2 Multinomial naive Bayes

Mô hình này chủ yếu được sử dụng trong phân loại văn bản mà vector đặc trưng được xây dựng dựa trên ý tưởng bag of words (BoW). Lúc này, mỗi văn bản được biểu diễn bởi một vector có độ dài d chính là số từ trong từ điển. Giá trị của thành phần thứ i trong mỗi vector chính là số lần từ thứ i xuất hiện trong văn bản đó. Khi đó, $p(x_i|c)$ tỉ lệ với tần suất từ thứ i (hay đặc trưng thứ i cho trường hợp tổng quát) xuất hiện trong các văn bản của class c . Giá trị này có thể được tính bằng

$$\lambda_{ci} = p(x_i|c) = \frac{N_{ci}}{N_c} \quad (11.7)$$

Trong đó:

- N_{ci} là tổng số lần từ thứ i xuất hiện trong các văn bản của class c . Nó chính là tổng của tất cả các đặc trưng thứ i của các vector đặc trưng ứng với class c .
- N_c là tổng số từ (kể cả lặp) xuất hiện trong class c . Nói cách khác, nó bằng tổng độ dài của toàn bộ các văn bản thuộc vào class c . Có thể suy ra rằng $N_c = \sum_{i=1}^d N_{ci}$, từ đó $\sum_{i=1}^d \lambda_{ci} = 1$. Ở đây d là số từ trong từ điển.

Cách tính này có một hạn chế là nếu có một từ mới chưa bao giờ xuất hiện trong class c thì biểu thức (11.7) sẽ bằng không, dẫn đến vế phải của (11.4) bằng không bất kể các giá trị còn lại có lớn thế nào (xem thêm ví dụ ở mục sau). Để giải quyết việc này, một kỹ thuật được gọi là *Laplace smoothing* được áp dụng:

$$\hat{\lambda}_{ci} = \frac{N_{ci} + \alpha}{N_c + d\alpha} \quad (11.8)$$

với α là một số dương, thường bằng 1, để tránh trường hợp tử số bằng không. Mẫu số được cộng với $d\alpha$ để đảm bảo tổng xác suất $\sum_{i=1}^d \hat{\lambda}_{ci} = 1$. Như vậy, mỗi class c sẽ được mô tả bởi một bộ các số dương có tổng bằng 1: $\hat{\lambda}_c = \{\hat{\lambda}_{c1}, \dots, \hat{\lambda}_{cd}\}$.

11.2.3 Bernoulli Naive Bayes

Mô hình này được áp dụng cho các loại dữ liệu mà mỗi thành phần là một giá trị nhị phân – bằng 0 hoặc 1. Ví dụ, cũng với loại văn bản nhưng thay vì đếm tổng số lần xuất hiện của một từ trong văn bản, ta chỉ cần quan tâm từ đó có xuất hiện hay không.

Khi đó, $p(x_i|c)$ được tính bằng:

$$p(x_i|c) = p(i|c)x_i + (1 - p(i|c))(1 - x_i) \quad (11.9)$$

với $p(i|c)$ có thể được hiểu là xác suất từ thứ i xuất hiện trong các văn bản của class c .

11.3 Ví dụ

11.3.1 Bắc hay Nam

Giả sử trong training set có các văn bản d1, d2, d3, d4 như trong Bảng 11.1. Mỗi văn bản này thuộc vào một trong hai lớp: B (*Bắc*) hoặc N (*Nam*). Hãy xác định lớp của văn bản d5.

Bảng 11.1: Ví dụ về nội dung của các văn bản trong bài toán Bắc hay Nam

	Văn bản	Nội dung	Lớp
Tập huấn luyện	d1	hanoi pho chaolong hanoi	B
	d2	hanoi buncha pho omai	B
	d3	pho banhgio omai	B
	d4	saigon hutiu banhbo pho	N
Kiểm thử	d5	hanoi hanoi buncha hutiu	?

Ta có thể dự đoán rằng d5 thuộc class *Bắc*.

Bài toán này có thể được giải quyết bằng NBC sử dụng multinomial Naive Bayes hoặc Bernoulli naive Bayes. Chúng ta sẽ cùng làm ví dụ với mô hình thứ nhất và triển khai code cho cả hai mô hình. Việc mô hình nào tốt hơn phụ thuộc vào mỗi bài toán. Chúng ta có thể thử cả hai để chọn ra mô hình tốt hơn.

Nhận thấy rằng ở đây có hai lớp B và N, ta cần đi tìm $p(B)$ và $p(N)$ dựa trên tần số xuất hiện của mỗi class trong tập training. Ta sẽ có

$$p(B) = \frac{3}{4}, \quad p(N) = \frac{1}{4} \quad (11.10)$$

Tập hợp toàn bộ các từ trong các văn bản, hay còn gọi là từ điển, là

$$V = \{\text{hanoi, pho, chaolong, buncha, omai, banhgio, saigon, hutiu, banhbo}\}$$

Tổng cộng số phần tử trong từ điển là $|V| = 9$.

Hình 11.1 minh họa quá trình huấn luyện và kiểm thử cho bài toán này khi sử dụng Multinomial naive Bayes, trong đó Laplace smoothing được sử dụng với $\alpha = 1$. Chú ý, hai giá trị tìm được 1.5×10^{-4} và 1.75×10^{-5} không phải là hai xác suất cần tìm mà chỉ là hai đại lượng **tỉ lệ thuận** với hai xác suất đó. Để tính cụ thể, ta có thể làm như sau

$$p(B|d5) = \frac{1.5 \times 10^{-4}}{1.5 \times 10^{-4} + 1.75 \times 10^{-5}} \approx 0.8955, \quad p(N|d5) = 1 - p(B|d5) \approx 0.1045 \quad (11.11)$$

Như vậy xác suất để d5 rơi vào class B là 89.55%, vào class N là 10.45%. Bạn đọc có thể tự tính với ví dụ khác: d6 = pho hutiu banhbo. Nếu tính toán đúng, ta sẽ thu được

$$p(B|d6) \approx 0.29, \quad p(N|d6) \approx 0.71 \quad (11.12)$$

và suy ra d6 thuộc vào class N.

Kết quả:

```
Predicting class of d5: B
Probability of d6 in each class: [[ 0.29175335  0.70824665]]
```

Kết quả này nhất quán với những kết quả được tính bằng tay ở trên.

Nếu sử dụng Bernoulli naive Bayes, chúng ta cần thay đổi một chút về feature vector. Lúc này, các giá trị khác không sẽ đều được đưa về 1 vì ta chỉ quan tâm đến việc từ đó có xuất hiện trong văn bản hay không.

```
from __future__ import print_function
from sklearn.naive_bayes import BernoulliNB
import numpy as np
# train data
d1 = [1, 1, 1, 0, 0, 0, 0, 0, 0]
d2 = [1, 1, 0, 1, 1, 0, 0, 0, 0]
d3 = [0, 1, 0, 0, 1, 1, 0, 0, 0]
d4 = [0, 1, 0, 0, 0, 0, 1, 1, 1]
train_data = np.array([d1, d2, d3, d4])
label = np.array(['B', 'B', 'B', 'N']) # 0 - B, 1 - N
# test data
d5 = np.array([[1, 0, 0, 1, 0, 0, 0, 1, 0]])
d6 = np.array([[0, 1, 0, 0, 0, 0, 0, 1, 1]])

## call MultinomialNB
model = BernoulliNB()
# training
model.fit(train_data, label)

# test
print('Predicting class of d5:', str(model.predict(d5)[0]))
print('Probability of d6 in each class:', model.predict_proba(d6))
```

Kết quả:

```
Predicting class of d5: B
Probability of d6 in each class: [[ 0.16948581  0.83051419]]
```

Ta thấy rằng, với bài toán nhỏ này, cả hai mô hình đều cho kết quả giống nhau (xác suất tìm được khác nhau nhưng không ảnh hưởng tới quyết định cuối cùng).

11.3.3 Naive Bayes Classifier cho bài toán spam filtering

Tiếp theo, chúng ta cùng làm việc với một bộ cơ sở dữ liệu lớn hơn. Dữ liệu trong ví dụ này được lấy trong *Exercise 6: Naive Bayes–Machine Learning*, Andrew Ng (<https://goo.gl/kbzR3d>). Trong ví dụ này, dữ liệu đã được xử lý, và là một tập con của cơ sở dữ liệu *Ling-Spam dataset* (<https://goo.gl/whHCd9>).

Mô tả dữ liệu Tập dữ liệu này bao gồm tổng cộng 960 email tiếng Anh, được tách thành training set và test set theo tỉ lệ 700:260 với 50% trong mỗi tập là các spam email.

Dữ liệu trong cơ sở dữ liệu này đã được xử lý khá đẹp. Các quy tắc xử lý như sau¹:

1. **Loại bỏ *stop words***: Những từ xuất hiện thường xuyên như ‘and’, ‘the’, ‘of’, v.v. được loại bỏ vì chúng xuất hiện ở cả hai loại, không ảnh hưởng nhiều đến việc quyết định.
2. **Lemmatization**: Những từ có cùng gốc được đưa về cùng loại. Ví dụ, ‘include’, ‘includes’, ‘included’ đều được đưa chung về ‘include’. Tất cả các từ cũng đã được đưa về dạng ký tự thường (không phải HOA).
3. **Loại bỏ *non-words***: các chữ số, dấu câu, và các ký tự đặc biệt đã được loại bỏ.

Dưới đây là một ví dụ của một email không phải spam, **trước khi được xử lý**.

Subject: Re: 5.1344 Native speaker intuitions

The discussion on native speaker intuitions has been extremely interesting, but I worry that my brief intervention may have muddied the waters. I take it that there are a number of separable issues. The first is the extent to which a native speaker is likely to judge a lexical string as grammatical or ungrammatical per se. The second is concerned with the relationships between syntax and interpretation (although even here the distinction may not be entirely clear cut).

và **sau khi được xử lý**:

re native speaker intuition discussion native speaker intuition extremely interest worry brief intervention muddy waters number separable issue first extent native speaker likely judge lexical string grammatical ungrammatical per se second concern relationship between syntax interpretation although even here distinction entirely clear cut

Và dưới đây là một ví dụ về **spam email sau khi được xử lý**.

financial freedom follow financial freedom work ethic extraordinary desire earn least per month work home special skills experience required train personal support need ensure success legitimate homebased income opportunity put back control finance life ve try opportunity past fail live promise

Chúng ta thấy rằng trong đoạn này có các từ như: *financial*, *extraordinary*, *earn*, *opportunity*, v.v. là những từ thường thấy trong các email spam.

Trong ví dụ này, chúng ta sẽ sử dụng Multinomial Naive Bayes.

¹ Bạn đọc có thể tham khảo thư viện NLTK (<http://www.nltk.org/>) cho các công việc xử lý dữ liệu này.

Để cho bài toán được đơn giản hơn, chúng ta sẽ tiếp tục sử dụng dữ liệu đã được xử lý, có thể được download tại <https://goo.gl/CSMxHU>. Trong folder sau khi giải nén, chúng ta sẽ thấy các file:

```
test-features.txt
test-labels.txt
train-features-50.txt
train-features-100.txt
train-features-400.txt
train-features.txt
train-labels-50.txt
train-labels-100.txt
train-labels-400.txt
train-labels.txt
```

tương ứng với các file chứa dữ liệu của training set và test set. File **train-features-50.txt** chứa dữ liệu của training set thu gọn với chỉ tổng cộng 50 email. Mỗi file **labels.txt** chứa nhiều dòng, mỗi dòng là một ký tự 0 hoặc 1 thể hiện email là *non-spam* hoặc *spam*.

Mỗi file **features.txt** chứa nhiều dòng, mỗi dòng có 3 số, chẳng hạn:

```
1 564 1
1 19 2
```

Trong đó, số đầu tiên là chỉ số của email, bắt đầu từ 1; số thứ hai là thứ tự của từ trong từ điển (tổng cộng 2500 từ); số thứ ba là số lượng của từ đó trong email đang xét. Dòng đầu tiên nói rằng trong email thứ nhất, từ thứ 564 trong từ điển xuất hiện một lần. Cách lưu dữ liệu như thế này giúp tiết kiệm bộ nhớ vì một email thường không chứa hết tất cả các từ trong từ điển mà chỉ chứa một lượng nhỏ, ta chỉ cần lưu các giá trị khác không.

Nếu biểu diễn mỗi email bằng một vector hàng có độ dài bằng độ dài từ điển (2500) thì dòng thứ nhất nói rằng đặc trưng thứ 564 của vector này bằng 1. Tương tự, đặc trưng thứ 19 của vector này bằng 2. Nếu không xuất hiện, các thành phần khác được mặc định bằng 0. Dựa trên các thông tin này, chúng ta có thể tiến hành lập trình với thư viện sklearn.

Khai báo thư viện và đường dẫn tới files:

```
from __future__ import print_function
import numpy as np
from scipy.sparse import coo_matrix # for sparse matrix
from sklearn.naive_bayes import MultinomialNB, BernoulliNB
from sklearn.metrics import accuracy_score # for evaluating results
# data path and file name
path = 'ex6DataPrepared/'
train_data_fn = 'train-features.txt'
test_data_fn = 'test-features.txt'
train_label_fn = 'train-labels.txt'
test_label_fn = 'test-labels.txt'
```

Tiếp theo ta cần viết hàm số đọc dữ liệu từ file `data_fn` với label tương ứng được lưu trong `label_fn`. Chú ý rằng số lượng từ trong từ điển là 2500.

Dữ liệu sẽ được lưu trong một ma trận mà mỗi hàng là một vector đặc trưng của email. Ma trận này là một ma trận sparse nên chúng ta sẽ sử dụng hàm `scipy.sparse.coo_matrix`.

```
nwords = 2500

def read_data(data_fn, label_fn):
    ## read label_fn
    with open(path + label_fn) as f:
        content = f.readlines()
        label = [int(x.strip()) for x in content]

    ## read data_fn
    with open(path + data_fn) as f:
        content = f.readlines()
    # remove '\n' at the end of each line
    content = [x.strip() for x in content]

    dat = np.zeros((len(content), 3), dtype = int)

    for i, line in enumerate(content):
        a = line.split(' ')
        dat[i, :] = np.array([int(a[0]), int(a[1]), int(a[2])])

    # remember to -1 at coordinate since we're in Python
    data = coo_matrix((dat[:, 2], (dat[:, 0] - 1, dat[:, 1] - 1)),\
        shape=(len(label), nwords))
    return (data, label)
```

Đoạn code dưới đây giúp lấy dữ liệu huấn luyện và kiểm thử, sau đó tiến hành phân lớp sử dụng `MultinomialNB`.

```
(train_data, train_label) = read_data(train_data_fn, train_label_fn)
(test_data, test_label) = read_data(test_data_fn, test_label_fn)

clf = MultinomialNB()
clf.fit(train_data, train_label)

y_pred = clf.predict(test_data)
print('Training size = %d, accuracy = %.2f%%' % \
    (train_data.shape[0], accuracy_score(test_label, y_pred)*100))
```

Kết quả:

```
Training size = 700, accuracy = 98.08%
```

Vậy là có tới 98.08% các email được phân loại đúng. Chúng ta tiếp tục thử với các bộ dữ liệu training nhỏ hơn.

```

train_data_fn = 'train-features-100.txt'
train_label_fn = 'train-labels-100.txt'
test_data_fn = 'test-features.txt'
test_label_fn = 'test-labels.txt'

(train_data, train_label) = read_data(train_data_fn, train_label_fn)
(test_data, test_label) = read_data(test_data_fn, test_label_fn)
clf = MultinomialNB()
clf.fit(train_data, train_label)
y_pred = clf.predict(test_data)
print('Training size = %d, accuracy = %.2f%%' % \
      (train_data.shape[0], accuracy_score(test_label, y_pred)*100))

train_data_fn = 'train-features-50.txt'
train_label_fn = 'train-labels-50.txt'
test_data_fn = 'test-features.txt'
test_label_fn = 'test-labels.txt'

(train_data, train_label) = read_data(train_data_fn, train_label_fn)
(test_data, test_label) = read_data(test_data_fn, test_label_fn)
clf = MultinomialNB()
clf.fit(train_data, train_label)
y_pred = clf.predict(test_data)
print('Training size = %d, accuracy = %.2f%%' % \
      (train_data.shape[0], accuracy_score(test_label, y_pred)*100))

```

Kết quả:

```

Training size = 100, accuracy = 97.69%
Training size = 50, accuracy = 97.31%

```

Ta thấy rằng thậm chí khi training set là rất nhỏ, 50 email tổng cộng, kết quả đạt được đã rất ấn tượng.

Nếu bạn muốn tiếp tục thử mô hình **BernoulliNB**:

```

clf = BernoulliNB(binarize = .5)
clf.fit(train_data, train_label)
y_pred = clf.predict(test_data)
print('Training size = %d, accuracy = %.2f%%' % \
      (train_data.shape[0], accuracy_score(test_label, y_pred)*100))

```

Kết quả:

```

Training size = 50, accuracy = 69.62%

```

Ta thấy rằng trong bài toán này, **MultinomialNB** hoạt động hiệu quả hơn.

11.4 Thảo luận

11.4.1 Tóm tắt

- Naive Bayes classifiers (NBC) thường được sử dụng trong các bài toán phân loại văn bản.
- NBC có thời gian huấn luyện và kiểm thử rất nhanh. Điều này có được là do giả sử về tính độc lập giữa các thành phần.
- Nếu giả sử về tính độc lập được thoả mãn (dựa vào bản chất của dữ liệu), NBC được cho là cho kết quả tốt hơn so với support vector machine (Phần VIII) và logistic regression (Chương 14) khi có ít dữ liệu huấn luyện.
- NBC có thể hoạt động với các vector đặc trưng mà một phần là liên tục (sử dụng Gaussian Naive Bayes), phần còn lại ở dạng rời rạc (sử dụng Multinomial hoặc Bernoulli). Chính sự độc lập giữa các đặc trưng khiến NBC có khả năng này.
- Khi sử dụng Multinomial Naive Bayes, Laplace smoothing thường được sử dụng để tránh trường hợp một từ trong dữ liệu kiểm thử chưa xuất hiện trong training set.
- Source code trong chương này có thể được tìm thấy [tại đây](#).

11.4.2 Đọc thêm

1. *Text Classification and Naive Bayes - Stanford* (<https://goo.gl/HcefLX>).
2. *6 Easy Steps to Learn Naive Bayes Algorithm (with code in Python)* (<https://goo.gl/odQaaY>).

Neural networks

Gradient descent

12.1 Giới thiệu

Hình 12.1 mô tả sự biến thiên của hàm số $f(x) = \frac{1}{2}(x - 1)^2 - 2$. Điểm màu xanh lục là một điểm cực tiểu (*local minimum*), và cũng là điểm làm cho hàm số đạt giá trị nhỏ nhất (*global minimum*). Global minimum là một trường hợp đặc biệt của local minimum.

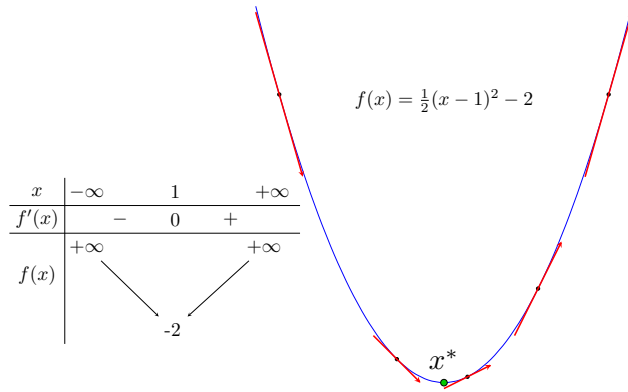
Giả sử ta đang quan tâm đến một hàm số một biến có đạo hàm mọi nơi. Cùng ôn lại một vài điểm cơ bản:

1. Điểm local minimum x^* của hàm số là điểm có đạo hàm $f'(x^*)$ bằng không. Hơn thế nữa, trong lân cận của nó, đạo hàm của các điểm phía bên trái x^* là không dương, đạo hàm của các điểm phía bên phải x^* là không âm.
2. Đường tiếp tuyến với đồ thị hàm số đó tại một điểm bất kỳ có hệ số góc chính bằng đạo hàm của hàm số tại điểm đó.

Trong Hình 12.1, các điểm bên trái của điểm local minimum màu xanh lục có đạo hàm âm, các điểm bên phải có đạo hàm dương. Và đối với hàm số này, càng xa về phía trái của điểm local minimum thì đạo hàm càng âm, càng xa về phía phải thì đạo hàm càng dương.

Trong machine learning nói riêng và toán tối ưu nói chung, chúng ta thường xuyên phải tìm các giá trị lớn nhất hoặc nhỏ nhất của một hàm số. Nếu chỉ xét riêng các hàm khả vi liên tục, việc giải phương trình đạo hàm bằng không thường rất phức tạp hoặc có thể ra vô số nghiệm. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là một nghiệm cần tìm của bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng không (ta vẫn đang giả sử rằng các hàm này liên tục và khả vi). Nếu bằng một cách nào đó có thể tìm được toàn bộ (hữu hạn) các điểm cực tiểu, ta chỉ cần thay từng điểm local minimum đó vào hàm số rồi tìm điểm làm cho hàm có giá trị nhỏ nhất. Tuy nhiên, trong hầu hết các trường hợp, việc



Hình 12.1: Khảo sát sự biến thiên của một đa thức bậc hai.

giải phương trình đạo hàm bằng không là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng của đạo hàm, từ việc các điểm dữ liệu có số chiều lớn, hoặc từ việc có quá nhiều điểm dữ liệu. Thực tế cho thấy, trong nhiều bài toán machine learning, các nghiệm local minimum thường đã cho kết quả tốt, đặc biệt là trong neural networks.

Hướng tiếp cận phổ biến nhất để giải quyết các bài toán tối ưu là xuất phát từ một điểm được coi là *gần* với nghiệm của bài toán, sau đó dùng một phép toán lặp để *tiến dần* đến điểm cần tìm, tức đến khi đạo hàm gần với không. Gradient descent (GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

12.2 GD cho hàm một biến

Xét các hàm số một biến $f : \mathbb{R} \rightarrow \mathbb{R}$. Quay trở lại Hình 12.1 và một vài quan sát đã nêu. Giả sử x_t là điểm tìm được sau vòng lặp thứ t . Ta cần tìm một thuật toán để đưa x_t về càng gần x^* càng tốt. Có hai quan sát sau đây:

1. Nếu đạo hàm của hàm số tại x_t là dương ($f'(x_t) > 0$) thì x_t nằm về bên phải so với x^* , và ngược lại. Để điểm tiếp theo x_{t+1} gần với x^* hơn, chúng ta cần di chuyển x_t về phía bên trái, tức về phía *âm*. Nói các khác, **ta cần di chuyển ngược dấu với đạo hàm**:

$$x_{t+1} = x_t + \Delta \quad (12.1)$$

Trong đó Δ là một đại lượng ngược dấu với đạo hàm $f'(x_t)$.

2. x_t càng xa x^* về phía bên phải thì $f'(x_t)$ càng lớn hơn 0 (và ngược lại). Vậy, lượng di chuyển Δ , một cách tự nhiên nhất, là tỉ lệ thuận với $-f'(x_t)$.

Hai nhận xét phía trên cho chúng ta một cách cập nhật đơn giản là

$$x_{t+1} = x_t - \eta f'(x_t) \quad (12.2)$$

Trong đó η là một số dương được gọi là *tốc độ học* (*learning rate*). Dấu trừ thể hiện việc chúng ta phải *đi ngược* với đạo hàm¹. Các quan sát đơn giản phía trên, mặc dù không phải đúng trong tất cả các trường hợp, là nền tảng cho rất nhiều phương pháp tối ưu.

¹ Đây chính là lý do phương pháp này được gọi là gradient descent—*descent* nghĩa là *đi ngược*

12.2.1 Ví dụ đơn giản với Python

Xét hàm số $f(x) = x^2 + 5 \sin(x)$ với đạo hàm $f'(x) = 2x + 5 \cos(x)$. Giả sử bắt đầu từ một điểm x_0 nào đó, tại vòng lặp thứ t , chúng ta sẽ cập nhật như sau:

$$x_{t+1} = x_t - \eta(2x_t + 5 \cos(x_t)) \quad (12.3)$$

Khi thực hiện trên Python, ta cần viết các hàm số²:

1. `grad` để tính đạo hàm.
2. `cost` để tính giá trị của hàm số. Hàm này không sử dụng trong thuật toán nhưng thường được dùng để kiểm tra việc tính đạo hàm có đúng không hoặc để xem giá trị của hàm số có giảm theo mỗi vòng lặp hay không.
3. `myGD1` là phần chính thực hiện thuật toán GD nêu phía trên. Đầu vào của hàm số này là learning rate và điểm xuất phát. Thuật toán dừng lại khi đạo hàm có độ lớn đủ nhỏ.

```
def grad(x):
    return 2*x + 5*np.cos(x)

def cost(x):
    return x**2 + 5*np.sin(x)

def myGD1(x0, eta):
    x = [x0]
    for it in range(100):
        x_new = x[-1] - eta*grad(x[-1])
        if abs(grad(x_new)) < 1e-3: # just a small number
            break
        x.append(x_new)
    return (x, it)
```

Điểm xuất phát khác nhau

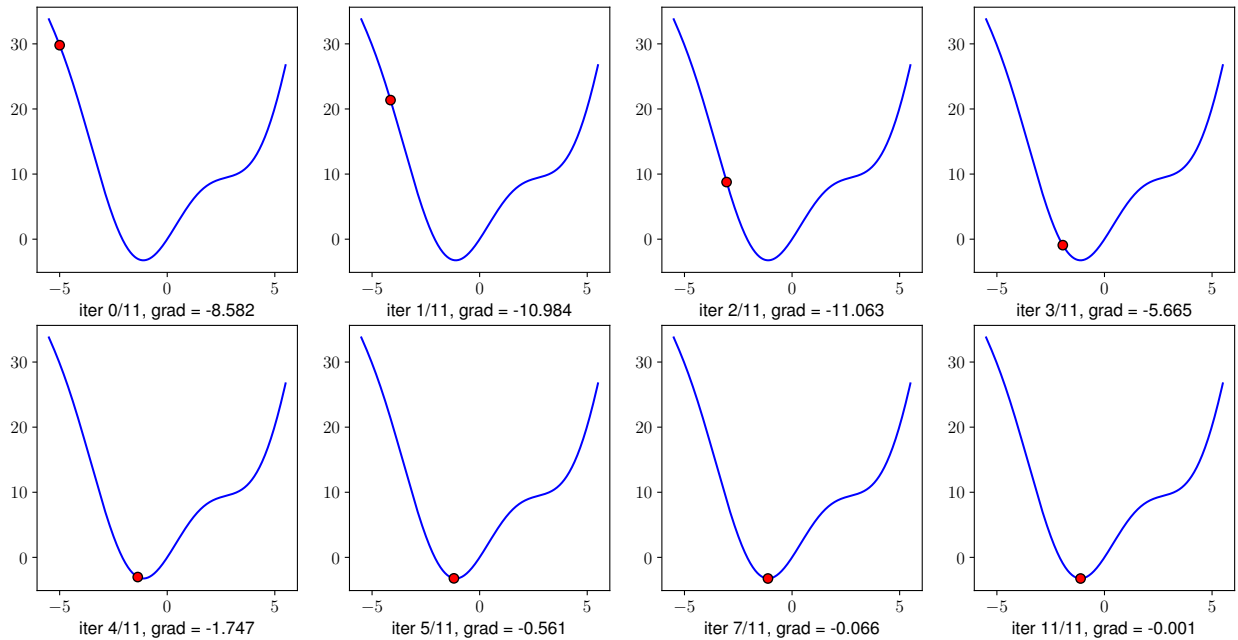
Sau khi đã có các hàm cần thiết, chúng ta thử tìm nghiệm với các điểm khởi tạo khác nhau là $x_0 = -5$ và $x_0 = 5$, với cùng learning rate $\eta = 0.1$.

```
(x1, it1) = myGD1(-5, .1)
(x2, it2) = myGD1(5, .1)
print('Solution x1 = %f, cost = %f, after %d iterations'%(x1[-1], cost(x1[-1]), it1))
print('Solution x2 = %f, cost = %f, after %d iterations'%(x2[-1], cost(x2[-1]), it2))
```

Kết quả:

```
Solution x1 = -1.110667, cost = -3.246394, after 11 iterations
Solution x2 = -1.110341, cost = -3.246394, after 29 iterations
```

² Giả sử rằng các thư viện đã được khai báo đầy đủ



Hình 12.2: Nghiệm tìm được qua các vòng lặp với $x_0 = -5, \eta = 0.1$

Vậy là với các điểm xuất phát khác nhau, thuật toán tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau. Hình 12.2 và Hình 12.3 thể hiện vị trí của nghiệm và đạo hàm qua các vòng lặp với cùng learning rate $\eta = .1$ nhưng điểm khởi tạo khác nhau tại -5 và 5 .

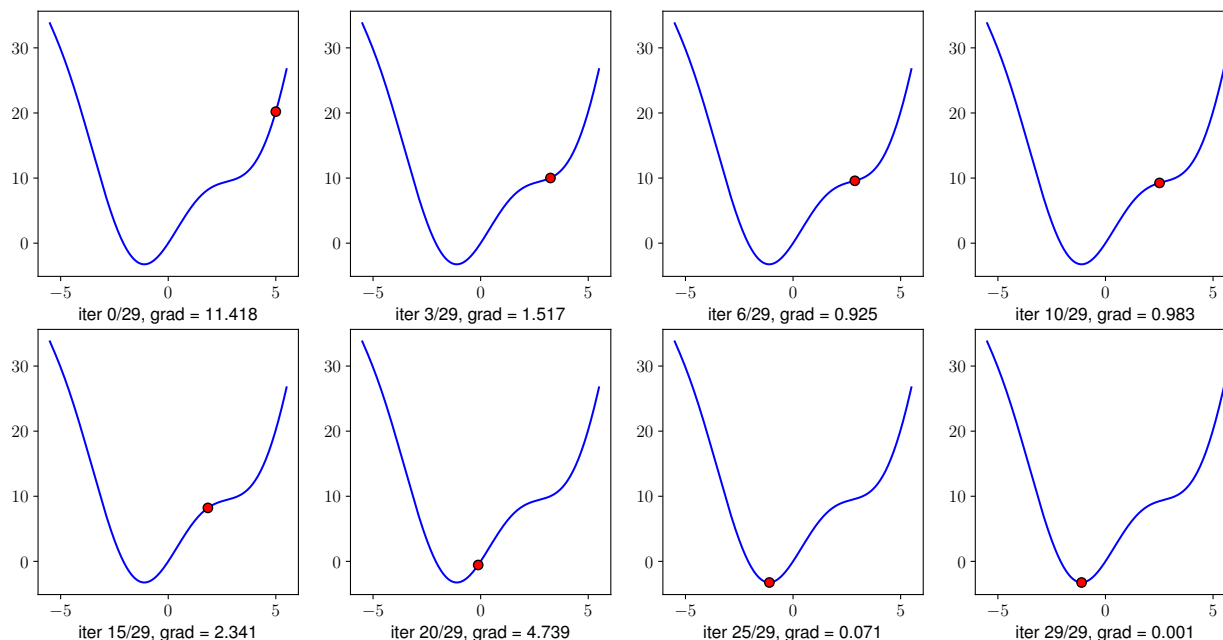
Hình 12.2 tương ứng với $x_0 = -5$, cho thấy nghiệm hội tụ nhanh hơn, vì điểm ban đầu x_0 gần với nghiệm $x^* \approx -1$ hơn. Hơn nữa, *đường đi* tới nghiệm khá suôn sẻ với đạo hàm luôn âm và càng gần nghiệm thì đạo hàm càng nhỏ.

Trong Hình 12.3 tương ứng với $x_0 = 5$, *đường đi* của nghiệm có chứa một khu vực có đạo hàm khá nhỏ gần điểm có hoành độ bằng 2.5. Điều này khiến cho thuật toán *la cà* ở đây khá lâu. Khi vượt qua được điểm này thì mọi việc diễn ra rất tốt đẹp. Các điểm không phải là điểm cực tiểu nhưng có đạo hàm gần bằng không rất dễ gây ra hiện tượng nghiệm bị *bẫy* (*trapped*) tại đây vì đạo hàm nhỏ khiến nó không thay đổi nhiều ở vòng lặp tiếp theo. Chúng ta sẽ thấy một kỹ thuật khác giúp *thoát* được những chiếc bẫy này.

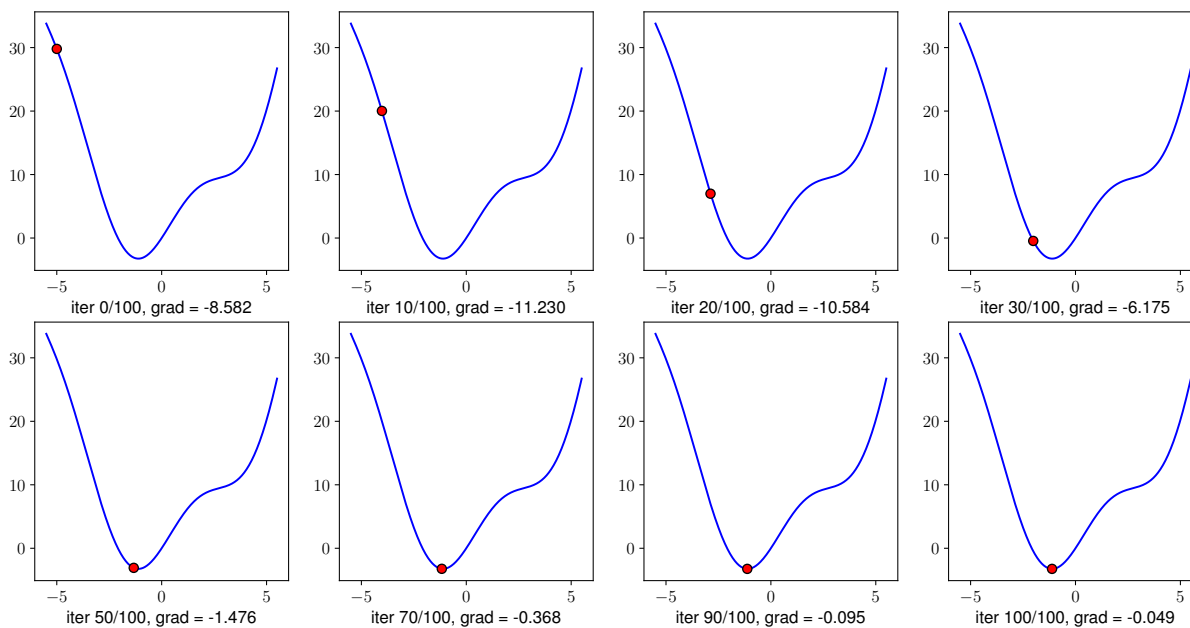
Learning rate khác nhau

Tốc độ hội tụ của GD không những phụ thuộc vào điểm xuất phát mà còn phụ thuộc vào *learning rate*. Hình 12.4 và Hình 12.5 thể hiện vị trí của nghiệm qua các vòng lặp với cùng điểm khởi tạo $x_0 = -5$ nhưng learning rate khác nhau. Ta quan sát thấy hai điều:

1. Với *learning rate* nhỏ $\eta = 0.01$ (Hình 12.4), tốc độ hội tụ rất chậm. Trong ví dụ này ta chọn tối đa 100 vòng lặp nên thuật toán dừng lại trước khi tới *đích*, mặc dù đã rất gần.

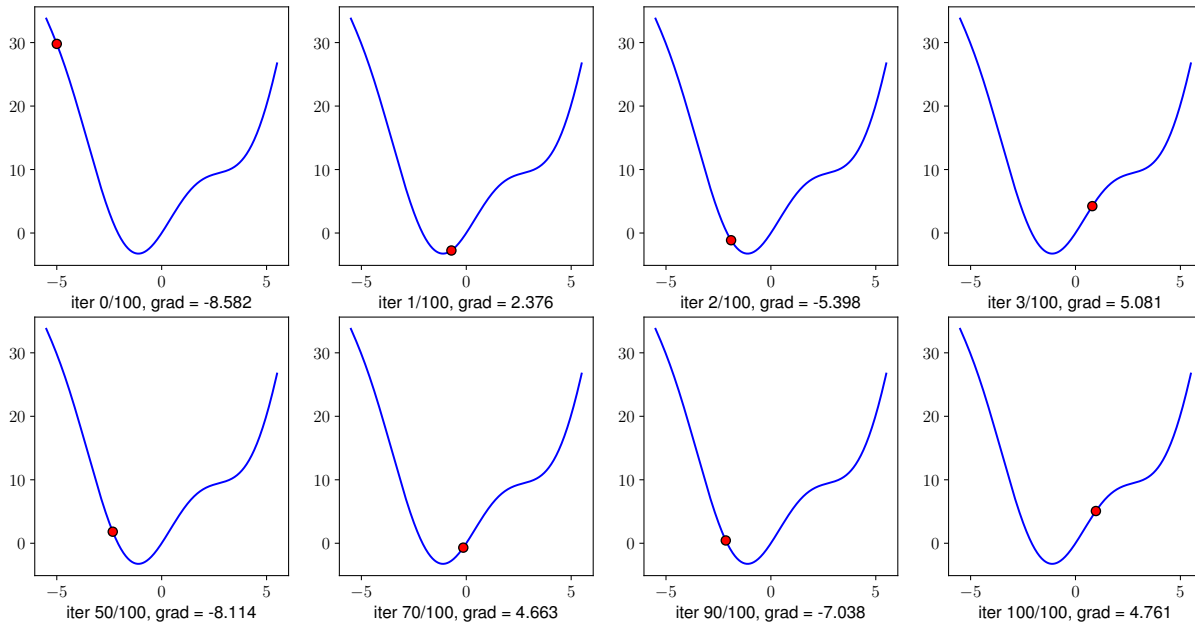


Hình 12.3: Nghiệm tìm được qua các vòng lặp với $x_0 = 5, \eta = 0.1$



Hình 12.4: Nghiệm tìm được qua các vòng lặp với điểm xuất phát $x_0 = -5$, learning rate $\eta = 0.01$.

Trong thực tế, khi việc tính toán trở nên phức tạp, *learning rate* quá thấp sẽ ảnh hưởng tới tốc độ của thuật toán rất nhiều, thậm chí không bao giờ tới được đích.



Hình 12.5: Nghiệm tìm được qua các vòng lặp với $x_0 = -5, \eta = 0.5$

2. Với *learning rate* lớn $\eta = 0.5$ (Hình 12.5), thuật toán tiến rất nhanh tới *gần đích* sau vài vòng lặp. Tuy nhiên, thuật toán không hội tụ được vì sự thay đổi vị trí của nghiệm sau mỗi vòng lặp là quá lớn, khiến nó cứ *quấn quanh* ở đích mà vẫn không tới được đích.

Việc lựa chọn *learning rate* rất quan trọng. Việc này phụ thuộc nhiều vào từng bài toán và phải làm một vài thí nghiệm để chọn ra giá trị tốt nhất. Ngoài ra, tùy vào một số bài toán, GD có thể làm việc hiệu quả hơn bằng cách chọn ra *learning rate* phù hợp hoặc chọn *learning rate* khác nhau ở mỗi vòng lặp, thường là giảm dần.

12.3 GD cho hàm nhiều biến

Giả sử ta cần tìm global minimum cho hàm $f(\theta)$ trong đó θ là tập hợp các tham số cần tối ưu. Đạo hàm của hàm số đó tại một điểm θ bất kỳ được ký hiệu là $\nabla_{\theta} f(\theta)$. Tương tự như hàm một biến, thuật toán GD cho hàm nhiều biến cũng bắt đầu bằng một điểm dự đoán θ_0 , sau đó, ở vòng lặp thứ t , quy tắc cập nhật là

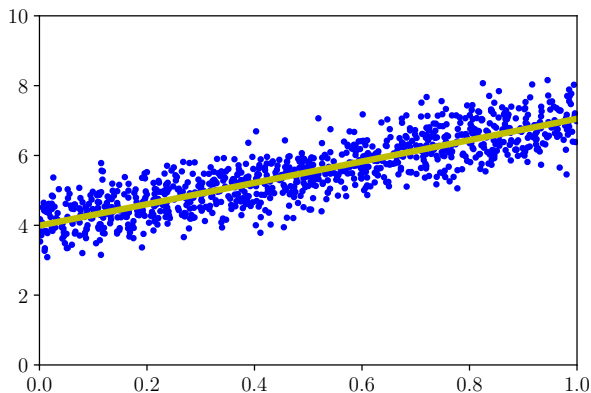
$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} f(\theta_t) \quad (12.4)$$

Hoặc viết dưới dạng đơn giản hơn: $\theta \leftarrow \theta - \eta \nabla_{\theta} f(\theta)$.

Quay lại với bài toán linear regression

Trong mục này, chúng ta quay lại với bài toán linear regression và thử tối ưu hàm mất mát của nó bằng thuật toán GD.

Nhắc lại hàm mất mát của linear regression và đạo hàm theo \mathbf{w} lần lượt là



Hình 12.6: Nghiệm của bài toán linear regression (đường thẳng màu vàng) tìm được bằng thư viện scikit-learn.

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \|\mathbf{y} - \mathbf{X}^T \mathbf{w}\|_2^2; \quad \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{N} \mathbf{X}(\mathbf{X}^T \mathbf{w} - \mathbf{y}) \quad (12.5)$$

Sau đây là ví dụ trên Python và một vài lưu ý khi lập trình

Trước hết, chúng ta tạo 1000 điểm dữ liệu được chọn *gần* với đường thẳng $y = 4 + 3x$:

```
from sklearn.linear_model import LinearRegression
X = np.random.rand(1000)
y = 4 + 3 * X + .5*np.random.randn(1000) # noise added
model = LinearRegression()
model.fit(X.reshape(-1, 1), y.reshape(-1, 1))
w, b = model.coef_[0][0], model.intercept_[0]
sol_sklearn = np.array([b, w])
print(sol_sklearn)
```

Kết quả:

```
Solution found by sklearn: [ 3.94323245  3.12067542]
```

Các điểm dữ liệu và đường thẳng tìm được bằng linear regression có phương trình $y \approx 3.94 + 3.12x$ được minh họa trong Hình 12.6. Nghiệm tìm được rất gần với mong đợi.

Tiếp theo, ta sẽ thực hiện tìm nghiệm của linear regression sử dụng GD. Ta cần viết hàm mất mát và đạo hàm theo \mathbf{w} . Chú ý rằng ở đây \mathbf{w} đã bao gồm cả bias.

```
def grad(w):
    N = Xbar.shape[0]
    return 1/N * Xbar.T.dot(Xbar.dot(w) - y)

def cost(w):
    N = Xbar.shape[0]
    return .5/N*np.linalg.norm(y - Xbar.dot(w))**2
```

Với các hàm phức tạp, khi tính xong đạo hàm chúng ta cần kiểm tra đạo hàm thông qua numerical gradient (xem Mục 2.6). Trường hợp này tương đối đơn giản, việc kiểm tra đạo hàm xin giành lại cho bạn đọc. Dưới đây là thuật toán GD cho bài toán.

```
def myGD(w_init, grad, eta):
    w = [w_init]
    for it in range(100):
        w_new = w[-1] - eta*grad(w[-1])
        if np.linalg.norm(grad(w_new))/len(w_new) < 1e-3:
            break
        w.append(w_new)
    return (w, it)

one = np.ones((X.shape[0],1))
Xbar = np.concatenate((one, X.reshape(-1, 1)), axis = 1)
w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
print('Sol found by GD: w = ', w1[-1].T, ', \nafter %d iterations.' %(it1+1))
```

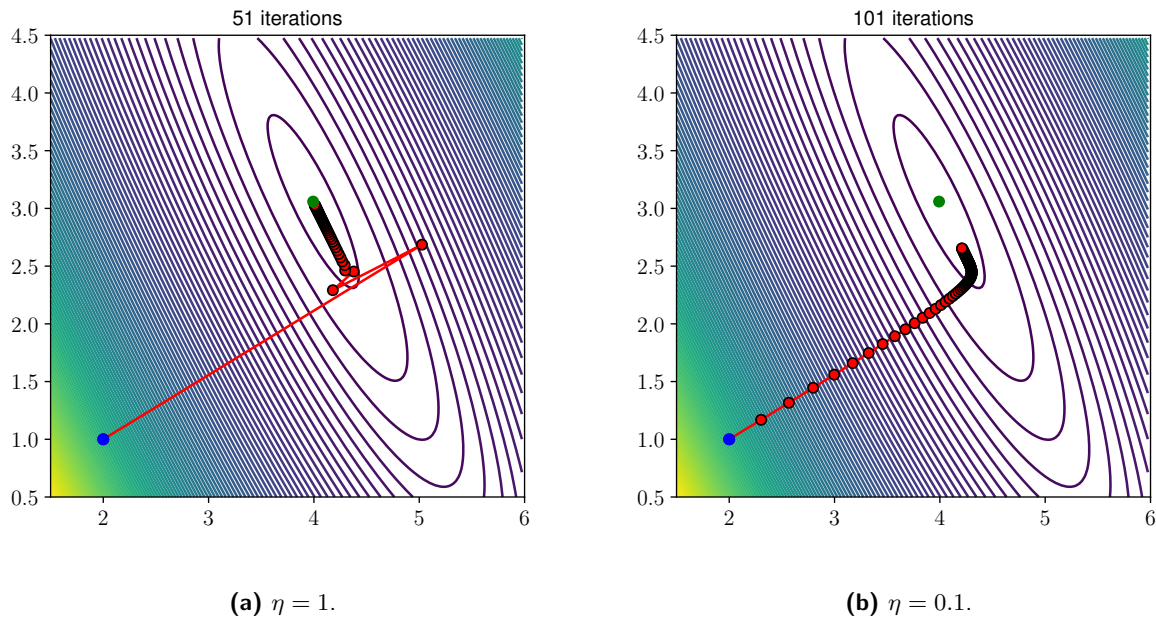
Kết quả:

```
Sol found by GD: w = [ 3.99026984  2.98702942] ,
after 49 iterations.
```

Sau 49 vòng lặp, thuật toán đã hội tụ với một nghiệm khá gần với nghiệm tìm được theo sklearn. Hình 12.7 mô tả đường đi của nghiệm với cùng điểm khởi tạo nhưng với learning rate khác nhau. Các điểm màu lam là các điểm xuất phát. Các điểm màu lục là nghiệm tìm được bằng thư viện scikit-learn. Các điểm màu đỏ là nghiệm qua các vòng lặp trung gian. Ta thấy rằng khi $\eta = 1$, thuật toán hội tụ tới (rất gần) nghiệm theo thư viện sau 49 vòng lặp. Với learning rate nhỏ hơn, $\eta = 0.1$, sau hơn 100 vòng lặp, nghiệm vẫn còn cách xa đích. Như vậy, việc chọn learning rate hợp lý là rất quan trọng.

Ở đây, chúng ta cùng làm quen với một khái niệm quan trọng: *đường đồng mức (level sets)*.

Ta thường gặp khái niệm *đường đồng mức* trong các bản đồ tự nhiên. Các điểm có cùng độ cao so với mực nước biển thường được nối với nhau. Với các ngọn núi, đường đồng mức thường là các đường kín bao quanh đỉnh núi. Khái niệm tương tự cũng được sử dụng trong tối ưu. *Đường đồng mức* hay *level sets* của một hàm số là tập hợp các điểm làm cho hàm số có cùng giá trị. Tưởng tượng một hàm số với hai biến, đồ thị của nó là một *bề mặt (surface)* trong không gian ba chiều. Đường đồng mức có thể được xác định bằng cách *cắt* bề mặt này bằng một mặt phẳng song song với đáy và lấy giao điểm của chúng. Với dữ liệu hai chiều, hàm mất mát của linear regression là một hàm bậc hai của hai thành phần trong vector hệ số \mathbf{w} . Đồ thị của nó là một bề mặt parabolic. Vì vậy, các đường đồng mức của hàm này là các đường ellipse có cùng tâm như trên Hình 12.7. Tâm này chính là đáy của parabolic và là giá trị nhỏ nhất của hàm mất mát. Các đường đồng mức được biểu diễn bởi các màu khác nhau với màu từ lam đậm đến lục, vàng, cam, đỏ, đỏ đậm thể hiện giá trị tăng dần.



Hình 12.7: Đường đi nghiệm của linear regression với các learning rate khác nhau.

12.4 GD với momentum

Trước hết, cùng nhắc lại thuật toán GD để tối ưu hàm mất mát $J(\theta)$:

1. Dự đoán một điểm khởi tạo $\theta = \theta_0$.
2. Cập nhật θ đến khi đạt được kết quả chấp nhận được:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \tag{12.6}$$

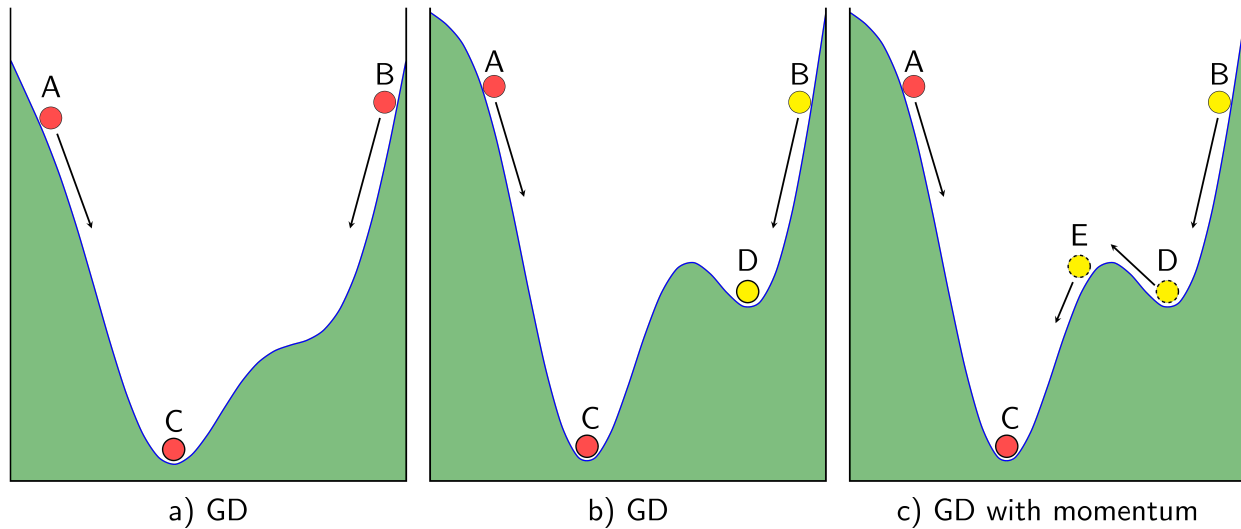
với $\nabla_{\theta} J(\theta)$ là đạo hàm của hàm mất mát tại θ .

Gradient dưới góc nhìn vật lý

Thuật toán GD thường được ví với tác dụng của trọng lực lên một hòn bi đặt trên một mặt có dạng một thung lũng như Hình 12.8a. Bất kể ta đặt hòn bi ở A hay B thì cuối cùng hòn bi cũng sẽ lăn xuống và kết thúc ở vị trí C.

Tuy nhiên, nếu như bề mặt có hai đáy thung lũng như Hình 12.8b thì tùy vào việc đặt bi ở A hoặc B, vị trí cuối cùng tương ứng của bi sẽ ở C hoặc D (giả sử rằng ma sát đủ lớn và đà chưa quá lớn khiến bi không thể vượt dốc). Điểm D chính là một điểm local minimum.

Nếu suy nghĩ một cách vật lý hơn, vẫn trong Hình 12.8b, nếu vận tốc ban đầu của bi khi ở điểm B đủ lớn, khi bi lăn đến điểm D, theo đà, bi có thể tiếp tục di chuyển lên dốc phía bên trái của D. Và nếu giả sử vận tốc ban đầu lớn hơn nữa, bi có thể vượt dốc tới điểm E rồi lăn



Hình 12.8: So sánh GD với các hiện tượng vật lý.

xuống C như trong Hình 12.8c. Dựa trên quan sát này, một thuật toán được ra đời nhằm giúp GD thoát được các local minimum. Thuật toán đó có tên là *momentum* (tức theo đà).

GD với momentum

Làm thế nào để biểu diễn *momentum* dưới dạng toán học?

Trong GD, chúng ta cần tính lượng thay đổi ở thời điểm t để cập nhật vị trí mới cho nghiệm (tức *hòn bi*). Nếu chúng ta coi đại lượng này như vận tốc v_t trong vật lý, vị trí mới của *hòn bi* sẽ là $\theta_{t+1} = \theta_t - v_t$, với giả sử rằng mỗi vòng lặp là một đơn vị thời gian. Dấu trừ thể hiện việc phải di chuyển ngược với đạo hàm. Việc tiếp theo là tính đại lượng v_t sao cho nó vừa mang thông tin của *độ dốc* (tức đạo hàm), vừa mang thông tin của *đà*, tức vận tốc trước đó v_{t-1} (với giả sử rằng vận tốc ban đầu $v_0 = 0$). Một cách đơn giản nhất, ta có thể lấy tổng có trọng số của chúng:

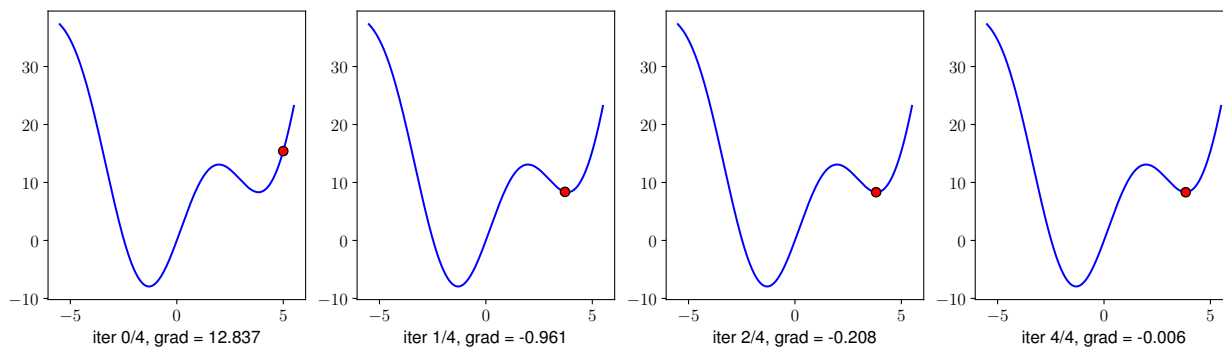
$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \quad (12.7)$$

Trong đó γ thường được chọn là một giá trị nhỏ hơn gần bằng một, thường là khoảng 0.9, v_{t-1} là vận tốc tại thời điểm trước đó, $\nabla_{\theta} J(\theta)$ chính là độ dốc của điểm trước đó. Sau đó, vị trí mới của *hòn bi* được xác định bởi

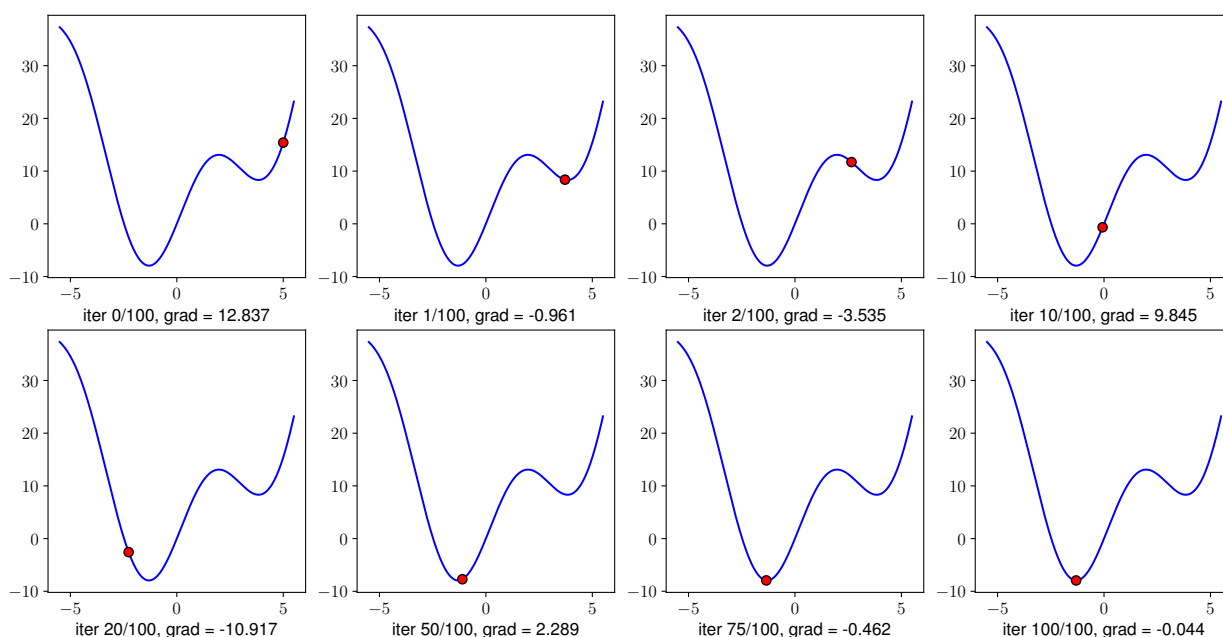
$$\theta \leftarrow \theta - v_t = \theta - \eta \nabla_{\theta} J(\theta) - \gamma v_{t-1} \quad (12.8)$$

Sự khác nhau giữa GD thông thường và GD với momentum chỉ nằm ở thành phần cuối cùng của (12.8). Thuật toán đơn giản này tỏ ra rất hiệu quả trong các bài toán thực tế. Dưới đây là một ví dụ trong không gian một chiều. Xét một hàm đơn giản có hai điểm local minimum, trong đó một điểm là global minimum

$$f(x) = x^2 + 10 \sin(x) \quad (12.9)$$



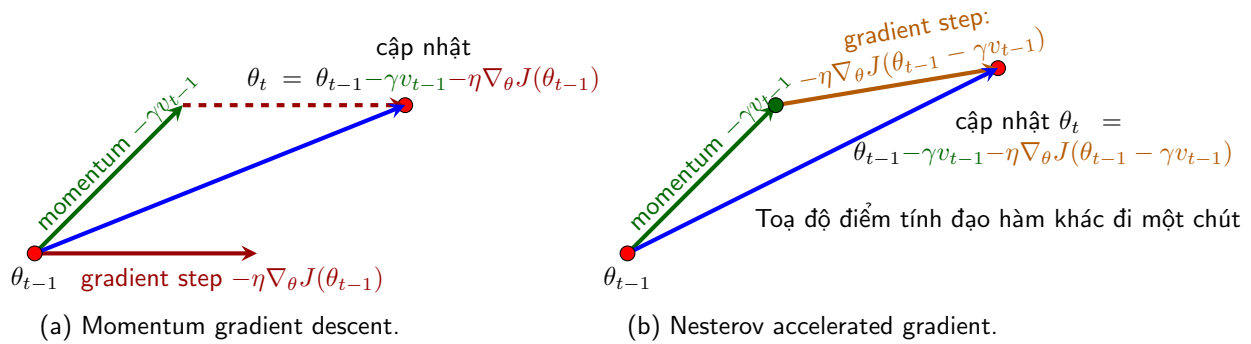
Hình 12.9: GD thông thường.



Hình 12.10: GD với momentum.

với đạo hàm $f'(x) = 2x + 10 \cos(x)$. Hình 12.9 thể hiện đường đi của nghiệm cho bài toán này khi không sử dụng momentum. Ta thấy rằng thuật toán hội tụ nhanh chóng sau chỉ bốn vòng lặp. Tuy nhiên, nghiệm dừng lại ở một điểm local minimum. Trong khi đó, Hình 12.10 thể hiện đường đi của nghiệm khi có sử dụng momentum. Chúng ta thấy rằng *hòn bi* vượt được dốc thứ nhất nhờ có *đà*, theo quán tính tiếp tục vượt qua điểm global minimum, nhưng quay trở lại điểm này sau 50 vòng lặp rồi chuyển động chậm dần quanh đó tới khi dừng hẳn ở vòng lặp thứ 100. Ví dụ này cho thấy momentum thực sự đã giúp nghiệm thoát được khu vực local minimum.

Nếu biết trước điểm *đặt bi* ban đầu θ , đạo hàm của hàm mất mát tại một điểm bất kỳ $\text{grad}(\theta)$, lượng thông tin lưu trữ từ vận tốc trước đó γ và learning rate η , chúng ta có thể viết hàm `GD_momentum` như sau.



Hình 12.11: Ý tưởng của Nesterov accelerated gradient.

```
def GD_momentum(grad, theta_init, eta, gamma):
    # Suppose we want to store history of theta
    theta = [theta_init]
    v_old = np.zeros_like(theta_init)
    for it in range(100):
        v_new = gamma*v_old + eta*grad(theta[-1])
        theta_new = theta[-1] - v_new
        if np.linalg.norm(grad(theta_new))/np.array(theta_init).size < 1e-3:
            break
        theta.append(theta_new)
        v_old = v_new
    return theta
```

12.5 Nesterov accelerated gradient

Momentum giúp *hòn bi* vượt qua được *dốc* local minimum. Tuy nhiên, có một hạn chế chúng ta có thể thấy trong ví dụ trên: khi tới gần *đích*, momentum vẫn mất khá nhiều thời gian trước khi dừng lại, cũng chính vì có *đà*. Một kỹ thuật có tên *Nesterov accelerated gradient* (NAG) [Nes07] giúp cho thuật toán momentum GD hội tụ nhanh hơn.

Ý tưởng chính

Ý tưởng trung tâm của thuật toán là *dự đoán vị trí của nghiệm trước một bước*. Cụ thể, nếu sử dụng số hạng *momentum* γv_{t-1} để cập nhật thì ta có thể *xấp xỉ* được vị trí tiếp theo của nghiệm là $\theta - \gamma v_{t-1}$. Vậy, thay vì sử dụng gradient của điểm hiện tại, NAG *đi trước một bước*, sử dụng gradient của điểm *được dự đoán* là vị trí tiếp theo. Ý tưởng này được thể hiện trên Hình 12.11.

- Với momentum thông thường, *lượng thay đổi* là tổng của hai vector: momentum vector và gradient ở thời điểm hiện tại.
- Với Nesterov momentum, *lượng thay đổi* là tổng của hai vector: momentum vector và gradient của điểm được dự đoán là vị trí tiếp theo.

Sự khác nhau giữa momentum và NAG nằm ở điểm lấy đạo hàm. Ở momentum, điểm được lấy đạo hàm chính là vị trí hiện tại của nghiệm. Ở NAG, điểm được lấy đạo hàm là điểm có được nếu sử dụng momentum.

Công thức cập nhật

Công thức cập nhật của NAG được cho như sau:

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \quad (12.10)$$

$$\theta \leftarrow \theta - v_t \quad (12.11)$$

Đoạn code dưới đây là hàm cho NAG.

```
def GD_NAG(grad, theta_init, eta, gamma):
    theta = [theta_init]
    v = [np.zeros_like(theta_init)]
    for it in range(100):
        v_new = gamma*v[-1] + eta*grad(theta[-1] - gamma*v[-1])
        theta_new = theta[-1] - v_new
        if np.linalg.norm(grad(theta_new))/np.array(theta_init).size < 1e-3:
            break
        theta.append(theta_new)
        v.append(v_new)
    return theta
```

Ví dụ minh họa

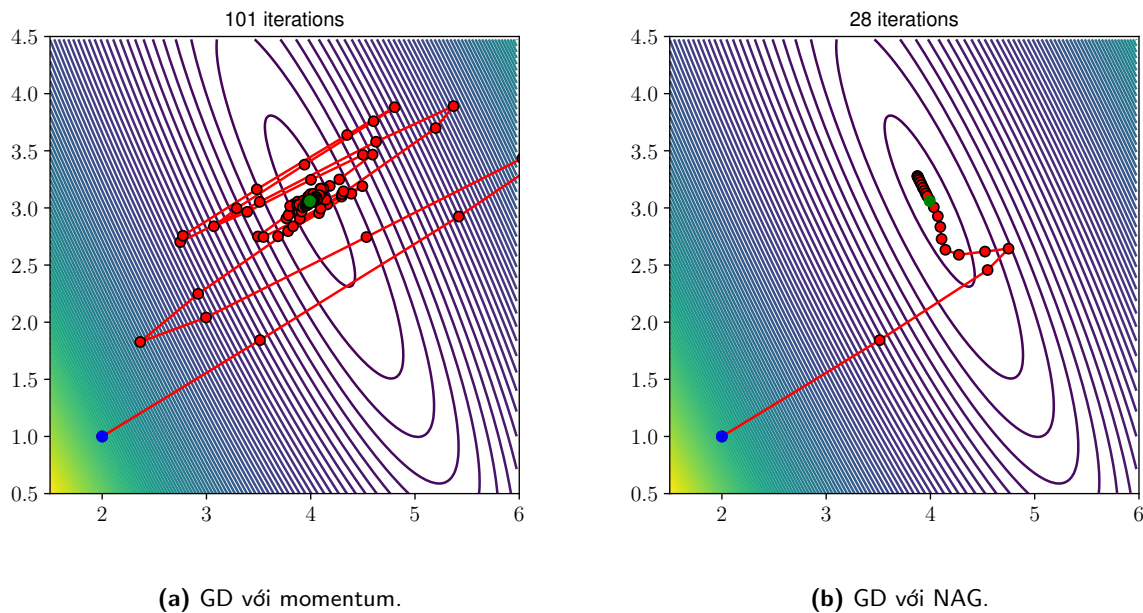
Chúng ta cùng áp dụng cả GD với momentum và GD với NAG cho bài toán linear regression đề cập ở trên. Hình 12.12 thể hiện đường đi của nghiệm khi sử dụng hai phương pháp này. Hình 12.12a là đường đi của nghiệm với phương pháp momentum. Nghiệm đi khá zigzag và mất nhiều vòng lặp hơn. Hình 12.12b là đường đi của nghiệm với phương pháp NAG, nghiệm hội tụ nhanh hơn, và đường đi ít zigzag hơn.

12.6 Stochastic gradient descent

12.6.1 Batch gradient descent

Thuật toán GD được đề cập từ đầu chương tới hiện tại còn được gọi là *batch GD*. Batch ở đây được hiểu là *tất cả*, tức khi cập nhật các tham số θ , chúng ta sử dụng **tất cả** các điểm dữ liệu \mathbf{x}_i . Hạn chế của việc này là khi lượng cơ sở dữ liệu lớn, có thể tới hàng triệu, việc tính toán đạo hàm trên toàn bộ dữ liệu tại mỗi vòng lặp sẽ tốn rất nhiều thời gian.

Online learning là khi cơ sở dữ liệu được cập nhật liên tục, mỗi lần tăng thêm vài điểm dữ liệu mới, yêu cầu cập nhật mô hình mới. Kéo theo đó là mô hình cũng phải được thay đổi một chút để phù hợp với dữ liệu mới. Nếu làm theo batch GD, tức tính lại đạo hàm của hàm mất mát tại tất cả các điểm dữ liệu, độ phức tạp tính toán sẽ rất cao. Lúc đó, thuật toán có thể không còn mang tính *online* nữa do mất quá nhiều thời gian tính toán.



Hình 12.12: Đường đi của nghiệm cho bài toán linear regression với hai phương pháp gradient descent khác nhau. NAG cho nghiệm mượt hơn và nhanh hơn.

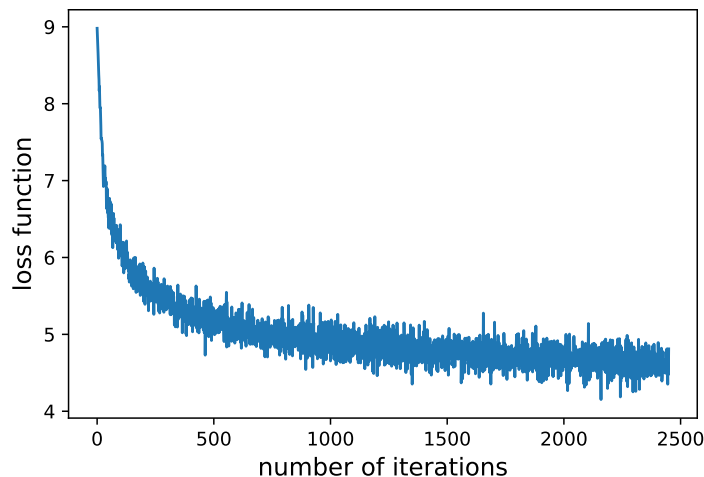
Một thuật toán đơn giản hơn, chấp nhận việc có sai số một chút nhưng lại lợi ích tính toán cao, thường được sử dụng có tên gọi là *stochastic gradient descent* (SGD).

12.6.2 Stochastic gradient descent

Trong SGD, tại một thời điểm (vòng lặp-iteration), ta chỉ tính đạo hàm của hàm mất mát dựa trên *chỉ một* điểm dữ liệu \mathbf{x}_i rồi cập nhật θ dựa trên đạo hàm này. Chú ý rằng hàm mất mát thường được lấy trung bình trên mỗi điểm dữ liệu nên đạo hàm tại một điểm cũng *được kỳ vọng* là khá gần với đạo hàm của hàm mất mát trên mọi điểm dữ liệu. Sau khi duyệt qua tất cả các điểm dữ liệu, thuật toán lặp lại quá trình trên. Biến thể đơn giản này trên thực tế làm việc rất hiệu quả.

epoch

Mỗi lần duyệt một lượt qua *tất cả* các điểm trên toàn bộ dữ liệu được gọi là một *epoch* (số nhiều *epoches*). Với GD thông thường, mỗi epoch ứng với một lần cập nhật θ . Với SGD, mỗi epoch ứng với N lần cập nhật θ với N là số điểm dữ liệu. Nhìn vào một mặt, việc cập nhật từng điểm một như thế này có thể làm giảm đi tốc độ thực hiện một epoch. Nhưng nhìn vào một mặt khác, với SGD, nghiệm có thể hội tụ sau vài epoch. Vì vậy, SGD phù hợp với các bài toán có lượng cơ sở dữ liệu lớn và các bài toán yêu cầu mô hình thay đổi liên tục như online learning. Với một mô hình đã được huấn luyện từ trước, khi có thêm dữ liệu, ta có thể chỉ cần chạy thêm một vài epoch nữa là đã có nghiệm hội tụ.



Hình 12.13: Ví dụ về giá trị hàm mất mát sau mỗi iteration khi sử dụng mini-batch gradient descent. Hàm mất mát *nhảy lên nhảy xuống* (*fluctuate*) sau mỗi lần cập nhật nhưng nhìn chung giảm dần và có xu hướng hội tụ.

Thứ tự lựa chọn điểm dữ liệu

Một điểm cần lưu ý đó là sau mỗi epoch, chúng ta cần *xáo trộn* (*shuffle*) thứ tự của các dữ liệu để đảm bảo tính ngẫu nhiên. Việc này cũng ảnh hưởng tới hiệu năng của SGD. Đây cũng chính là lý do thuật toán này có chứa từ *stochastic* (*ngẫu nhiên*).

Một cách toán học, quy tắc cập nhật của SGD là

$$\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta; \mathbf{x}_i, \mathbf{y}_i) \quad (12.12)$$

trong đó $J(\theta; \mathbf{x}_i, \mathbf{y}_i) \triangleq J_i(\theta)$ là hàm mất mát với chỉ một điểm dữ liệu thứ i . Các thuật toán biến thể của GD như momentum hay NAG hoàn toàn có thể được áp dụng vào SGD.

12.6.3 Mini-batch gradient descent

Khác với SGD, mini-batch sử dụng một số lượng k lớn hơn một (nhưng vẫn nhỏ hơn tổng số điểm dữ liệu N rất nhiều) để cập nhật ở mỗi *iteration*. Giống với SGD, mini-batch GD bắt đầu mỗi epoch bằng việc xáo trộn ngẫu nhiên dữ liệu rồi chia toàn bộ dữ liệu thành các *mini-batch*, mỗi *mini-batch* có k điểm dữ liệu (trừ mini-batch cuối có thể có ít hơn nếu N không chia hết cho k). Ở mỗi vòng lặp, thuật toán này lấy ra một mini-batch để tính toán đạo hàm rồi cập nhật. Một epoch cũng là khi thuật toán chạy hết dữ liệu một lượt. Như vậy, *một epoch* bao gồm xấp xỉ N/k lần *iteration*. Mini-batch GD được sử dụng trong hầu hết các thuật toán machine learning, đặc biệt là trong deep learning. Giá trị k được gọi là *batch size* (không phải *mini-batch size*) thường được chọn là khoảng từ vài chục đến vài trăm.

Hình 12.13 là ví dụ về giá trị của hàm mất mát của một bài toán khác phức tạp hơn mỗi khi cập nhật tham số θ khi sử dụng mini-batch gradient descent. Mặc dù giá trị của hàm mất mát sau các vòng lặp không phải lúc nào cũng giảm, chúng ta vẫn thấy rằng giá trị này có xu hướng giảm và hội tụ.

12.7 Thảo luận

12.7.1 Điều kiện dừng thuật toán

Có một điểm chúng ta chưa đề cập kỹ-khi nào thì nên dừng thuật toán gradient descent?

Trong thực nghiệm, chúng ta có thể kết hợp các phương pháp sau.

1. Giới hạn số vòng lặp. Một nhược điểm của cách làm này là có thể thuật toán dừng lại trước khi nghiệm đủ tốt. Tuy nhiên, đây là phương pháp phổ biến nhất và cũng để đảm bảo rằng chương trình chạy không quá lâu.
2. So sánh gradient của nghiệm tại hai lần cập nhật liên tiếp, khi nào giá trị này đủ nhỏ thì dừng lại. Phương pháp này cũng có một nhược điểm lớn là việc tính đạo hàm đôi khi trở nên quá phức tạp.
3. So sánh giá trị của hàm mất mát của nghiệm tại hai lần cập nhật liên tiếp, khi nào giá trị này đủ nhỏ thì dừng lại. Nhược điểm của phương pháp này là nếu tại một thời điểm, đồ thị hàm số có dạng *bằng phẳng* tại một khu vực nhưng khu vực đó không chứa điểm local minimum, thuật toán cũng dừng lại trước khi đạt giá trị mong muốn.
4. Vừa chạy gradient descent, vừa kiểm tra kết quả. Một kỹ thuật thường được sử dụng nữa là cho thuật toán chạy với số lượng vòng lặp cực lớn. Trong quá trình chạy, chương trình thường xuyên kiểm tra chất lượng mô hình bằng cách áp dụng nó lên dữ liệu tập huấn luyện và/hoặc validation. Đồng thời, mô hình sau một vài vòng lặp được lưu lại trong bộ nhớ. Mô hình tốt nhất có thể không phải là mô hình với số vòng lặp lớn hơn.

12.7.2 Đọc thêm

Source code trong chương này có thể được tìm thấy tại <https://goo.gl/RJrRv7>.

Ngoài các thuật toán đã đề cập trong chương này, rất nhiều thuật toán khác giúp cải thiện gradient descent được đề xuất gần đây [Rud16]. Bạn đọc có thể đọc thêm AdaGrad [DHS11], RMSProp [TH12], Adam [KB14], v.v..

Các trang web và video dưới đây cũng là các tài liệu tốt cho gradient descent.

1. *An overview of gradient descent optimization algorithms* (<https://goo.gl/AGwbbg>).
2. *Stochastic Gradient descent-Wikipedia* (<https://goo.gl/pmuLzk>).
3. *Stochastic gradient descent-Andrew Ng* (<https://goo.gl/jgBf2N>).
4. *An Interactive Tutorial on Numerical Optimization* (<https://goo.gl/t85mvA>).
5. *Machine Learning cơ bản, Bài 7, 8* (<https://goo.gl/US17PP>).

Perceptron learning algorithm

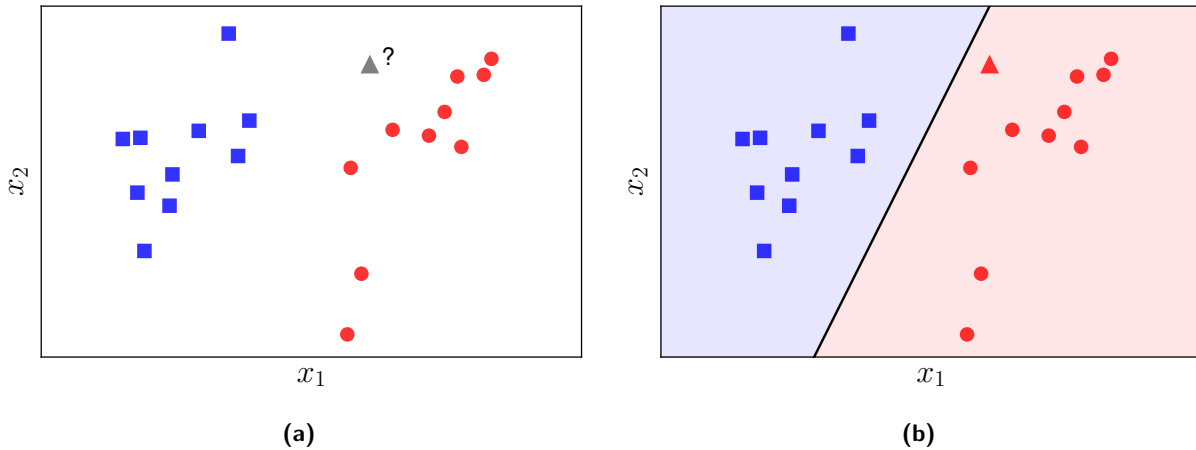
13.1 Giới thiệu

Trong chương này, chúng ta cùng tìm hiểu một trong các thuật toán đầu tiên trong lịch sử machine learning. Đây là một thuật toán phân lớp đơn giản có tên là *perceptron learning algorithm* (PLA [Ros57]). Thuật toán này được thiết kế cho bài toán *phân lớp nhị phân* (*binary classification*) với chỉ hai lớp dữ liệu. Đây là nền tảng cho các thuật toán liên quan tới neural networks rồi deep learning sau này.

Giả sử có hai class đã được gán nhãn được minh họa trong Hình 13.1a tương ứng với tập các điểm màu xanh và tập các điểm màu đỏ. Bài toán đặt ra là từ dữ liệu của hai tập được gán nhãn cho trước, hãy xây dựng một bộ phân lớp có khả năng dự đoán được nhãn (màu) của một điểm dữ liệu mới, chẳng hạn điểm màu xám.

Nếu coi mỗi vector đặc trưng là một điểm trong không gian nhiều chiều, bài toán phân lớp có thể được coi như bài toán xác định mỗi điểm trong không gian thuộc vào lớp nào. Nói cách khác, nếu ta coi mỗi lớp *chiếm* một hoặc vài vùng *lãnh thổ* trong không gian, ta cần đi tìm *ranh giới* (*boundary*) giữa các vùng đó. Ranh giới đơn giản nhất trong không gian hai chiều là một đường thẳng, trong không gian ba chiều là một mặt phẳng, trong không gian nhiều chiều hơn là một *siêu mặt phẳng* hoặc *siêu phẳng* (*hyperplane*). Những ranh giới phẳng này được coi là đơn giản vì chúng có thể được biểu diễn dưới dạng toán học bằng một hàm số tuyến tính. Tất nhiên, ta đang giả sử rằng tồn tại một siêu phẳng như vậy. Hình 13.1b minh họa một đường thẳng phân chia hai lớp trong không gian hai chiều. Lãnh thổ của hai lớp xanh và đỏ được mô tả bởi hai nửa mặt phẳng với màu tương ứng. Trong trường hợp này, điểm dữ liệu mới hình tam giác được phân vào lớp đỏ.

Perceptron learning algorithm (PLA) là một thuật toán đơn giản giúp tìm một ranh giới siêu phẳng cho bài toán phân lớp nhị phân, với giả sử rằng tồn tại ranh giới phẳng đó. Nếu hai lớp dữ liệu có thể được phân chia hoàn toàn bằng một siêu phẳng, ta nói rằng hai lớp đó *linearly separable*.



Hình 13.1: Bài toán phân lớp nhị phân trong không gian hai chiều. (a) Cho hai lớp dữ liệu vuông xanh và tròn đỏ, hãy xác định điểm dữ liệu tam giác xám thuộc lớp nào. (b) Ví dụ về một ranh giới phẳng của hai lớp, điểm tam giác được phân vào lớp đỏ với đường ranh giới này.

13.2 Thuật toán perceptron

13.2.1 Cách phân lớp của perceptron learning algorithm

Giả sử $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ là ma trận chứa các điểm dữ liệu huấn luyện mà mỗi cột \mathbf{x}_i là một điểm dữ liệu trong không gian d chiều. Giả sử thêm các nhãn tương ứng với từng điểm dữ liệu được lưu trong một vector hàng $\mathbf{y} = [y_1, y_2, \dots, y_N] \in \mathbb{R}^{1 \times N}$, với $y_i = 1$ nếu \mathbf{x}_i thuộc lớp thứ nhất (vuông xanh) và $y_i = -1$ nếu \mathbf{x}_i thuộc lớp còn lại (tròn đỏ).

Tại một thời điểm, giả sử ta tìm được ranh giới là một siêu phẳng có phương trình

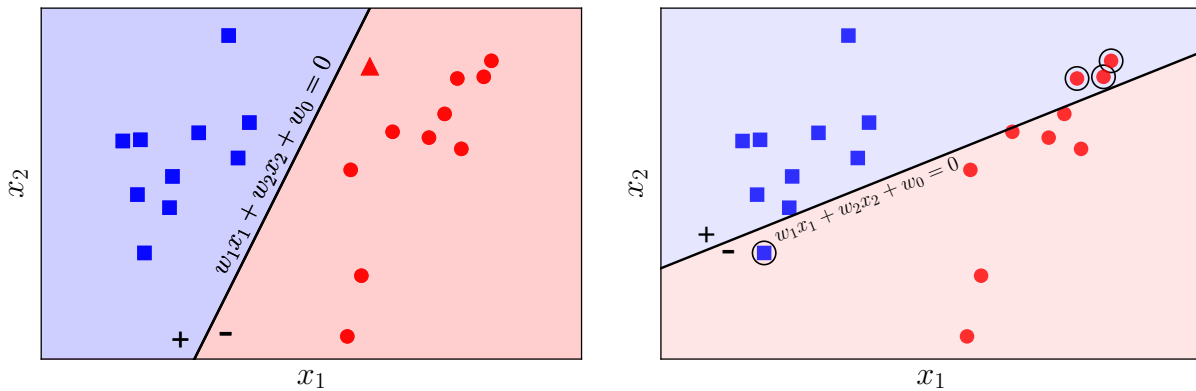
$$f_{\mathbf{w}}(\mathbf{x}) = w_1x_1 + \dots + w_dx_d + w_0 = \mathbf{w}^T \mathbf{x} + w_0 = 0 \quad (13.1)$$

với $\mathbf{w} \in \mathbb{R}^d$ là vector hệ số và w_0 là số hạng tự do được gọi là bias. Bằng cách sử dụng bias trick (xem Mục 7.2.4), ta có thể coi phương trình siêu phẳng là $f_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} = 0$ với \mathbf{x} ở đây được ngầm hiểu là vector đặc trưng mở rộng thêm một đặc trưng bằng 1. Vector hệ số \mathbf{w} cũng chính là *vector pháp tuyến* của siêu phẳng $\mathbf{w}^T \mathbf{x} = 0$.

Trong không gian hai chiều, giả sử đường thẳng $w_1x_1 + w_2x_2 + w_0 = 0$ chính là nghiệm cần tìm như Hình 13.2a. Nhận xét rằng các điểm nằm về cùng một phía so với đường thẳng này sẽ làm cho hàm số $f_{\mathbf{w}}(\mathbf{x})$ mang cùng dấu. Chỉ cần đổi dấu của \mathbf{w} nếu cần thiết, ta có thể giả sử các điểm nằm trong nửa mặt phẳng nền xanh mang dấu dương (+), các điểm nằm trong nửa mặt phẳng nền đỏ mang dấu âm (-). Các dấu này cũng tương đương với nhãn y của mỗi nhãn. Vậy nếu \mathbf{w} là một nghiệm của bài toán perceptron, với một điểm dữ liệu mới \mathbf{x} chưa được gán nhãn, ta có thể xác định nhãn của nó bằng một phép toán đơn giản:

$$\text{label}(\mathbf{x}) = \begin{cases} 1 & \text{nếu } \mathbf{w}^T \mathbf{x} \geq 0 \\ -1 & \text{o.w.} \end{cases} \quad (13.2)$$

Nói cách khác, $\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x})$ với sgn là hàm xác định dấu, giả sử rằng $\text{sgn}(0) = 1$.



(a) Đường thẳng phân chia không có lỗi. (b) Đường thẳng phân chia có lỗi tại các điểm khoanh tròn.

Hình 13.2: Ví dụ về các đường thẳng trong không gian hai chiều: (a) một nghiệm của bài toán PLA, (b) không phải nghiệm.

13.2.2 Xây dựng hàm mất mát

Tiếp theo, chúng ta xây dựng một hàm mất mát với tham số \mathbf{w} bất kỳ. Vẫn trong không gian hai chiều, giả sử đường thẳng $w_1x_1 + w_2x_2 + w_0 = 0$ được cho như Hình 13.2b. Các điểm được khoanh tròn là các điểm bị *phân lớp lỗi* (*misclassified*). Ta luôn muốn rằng không có điểm nào bị phân lớp lỗi. Một cách tự nhiên, ta có thể sử dụng hàm *đếm số lượng* các điểm bị phân lớp lỗi và tìm cách tối thiểu hàm số này.

Xét một điểm \mathbf{x}_i bất kỳ với nhãn y_i . Nếu nó bị phân lớp lỗi, ta phải có $\text{sgn}(\mathbf{w}^T \mathbf{x}_i) \neq y_i$. Vì hai giá trị này chỉ bằng 1 hoặc -1 , ta sẽ có $y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i) = -1$. Như vậy, hàm số đếm số lượng điểm bị phân lớp lỗi có thể được viết dưới dạng

$$J_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \text{sgn}(\mathbf{w}^T \mathbf{x}_i)) \tag{13.3}$$

trong đó \mathcal{M} ký hiệu tập các điểm bị phân lớp lỗi ứng với mỗi \mathbf{w} . Mục đích cuối cùng là đi tìm \mathbf{w} sao cho không có điểm nào bị phân lớp lỗi, tức $J_1(\mathbf{w}) = 0$. Một điểm quan trọng, đây là một hàm số rời rạc nên rất khó được tối ưu. Chúng ta cần tìm một hàm mất mát khác để việc tối ưu khả thi hơn. Xét hàm mất mát

$$J(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T \mathbf{x}_i) \tag{13.4}$$

Hàm $J(\mathbf{w})$ khác một chút với hàm $J_1(\mathbf{w})$ ở chỗ hàm rời rạc sgn đã được lược bỏ. Ngoài ra, khi một điểm bị phân lớp lỗi \mathbf{x}_i nằm càng xa ranh giới, giá trị $-y_i \mathbf{w}^T \mathbf{x}_i$ sẽ càng lớn, nghĩa là hàm mất mát sẽ lớn lên. Vì tổng vẫn được tính trên các tập điểm bị phân lớp lỗi \mathcal{M} , giá trị nhỏ nhất của hàm mất mát này cũng bằng không nếu không có điểm nào bị phân lớp lỗi. Vì vậy, $J(\mathbf{w})$ được cho là tốt hơn $J_1(\mathbf{w})$ vì nó *trừng phạt* rất nặng những điểm *lấn sâu sang lãnh thổ của lớp kia*. Trong khi đó, $J_1(\cdot)$ *trừng phạt* các điểm phân lớp lỗi một lượng như nhau bằng một, bất kể chúng gần hay xa ranh giới.

13.2.3 Tối ưu hàm mất mát

Tại một thời điểm, nếu ta chỉ quan tâm tới các điểm bị phân lớp lỗi thì hàm số $J(\mathbf{w})$ khả vi tại mọi \mathbf{w} , vậy ta có thể sử dụng gradient descent hoặc stochastic gradient descent (SGD) để tối ưu hàm mất mát này. Chúng ta sẽ giải quyết bài toán tối ưu hàm mất mát $J(\mathbf{w})$ bằng SGD bằng cách cập nhật \mathbf{w} tại mỗi vòng lặp dựa trên chỉ một điểm dữ liệu. Với chỉ một điểm dữ liệu \mathbf{x}_i bị phân lớp lỗi, hàm mất mát và đạo hàm của nó lần lượt là

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{w}^T \mathbf{x}_i; \quad \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{x}_i \quad (13.5)$$

Vậy quy tắc cập nhật \mathbf{w} sử dụng SGD là

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(-y_i \mathbf{x}_i) = \mathbf{w} + \eta y_i \mathbf{x}_i \quad (13.6)$$

với η là learning rate. Trong PLA, η được chọn bằng 1. Ta có một quy tắc cập nhật rất gọn:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i \quad (13.7)$$

Nói cách khác, với mỗi điểm \mathbf{x}_i bị phân lớp lỗi, bằng cách nhân điểm đó với nhãn y_i của nó, lấy kết quả cộng vào \mathbf{w} hiện tại, ta sẽ được \mathbf{w} mới. Tiếp theo, ta thấy rằng

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = (\mathbf{w}_t + y_i \mathbf{x}_i)^T \mathbf{x}_i = \mathbf{w}_t^T \mathbf{x}_i + y_i \|\mathbf{x}_i\|_2^2 \quad (13.8)$$

Nếu $y_i = 1$, vì \mathbf{x}_i bị phân lớp lỗi nên $\mathbf{w}_t^T \mathbf{x}_i < 0$. Cũng vì $y_i = 1$ nên $y_i \|\mathbf{x}_i\|_2^2 = \|\mathbf{x}_i\|_2^2 \geq 1$ (chú ý \mathbf{x}_i là một vector đặc trưng mở rộng với một phần tử bằng 1). Từ đó suy ra $\mathbf{w}_{t+1}^T \mathbf{x}_i > \mathbf{w}_t^T \mathbf{x}_i$. Nói cách khác, $-y_i \mathbf{w}_{t+1}^T \mathbf{x}_i < -y_i \mathbf{w}_t^T \mathbf{x}_i$. Điều tương tự cũng xảy ra với $y_i = -1$. Việc này chỉ ra rằng đường thẳng được mô tả bởi \mathbf{w}_{t+1} có xu hướng khiến hàm mất mát tại điểm bị phân lớp lỗi \mathbf{x}_i giảm đi. *Chú ý rằng việc này không đảm bảo hàm mất mát trên toàn bộ dữ liệu sẽ giảm, vì rất có thể đường thẳng mới sẽ làm cho một điểm lúc trước được phân lớp đúng trở thành một điểm bị phân lớp sai. Tuy nhiên, thuật toán này được đảm bảo sẽ hội tụ sau một số hữu hạn bước.* Thuật toán perceptron được tóm tắt dưới đây.

Thuật toán 13.1: Perceptron

1. Tại thời điểm $t = 0$, chọn ngẫu nhiên một vector hệ số \mathbf{w}_0 .
2. Tại thời điểm t , nếu không có điểm dữ liệu nào bị phân lớp lỗi, dừng thuật toán.
3. Giả sử \mathbf{x}_i là một điểm bị phân lớp lỗi. Cập nhật

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$$

4. Thay đổi $t = t + 1$ rồi quay lại Bước 2.

13.2.4 Chứng minh hội tụ

Gọi \mathbf{w}^* là một nghiệm của bài toán phân lớp nhị phân với hai lớp linearly separable. Nghiệm này luôn tồn tại khi hai lớp là linearly separable. Ta sẽ chứng minh Thuật toán 13.1 kết thúc sau một số hữu hạn bước bằng phản chứng.

Giả sử ngược lại, tồn tại một \mathbf{w} mà Thuật toán 13.1 chạy mãi mãi. Trước hết ta thấy rằng, với $\alpha > 0$ bất kỳ, nếu \mathbf{w}^* là nghiệm, $\alpha\mathbf{w}^*$ cũng là nghiệm của bài toán. Xét dãy số không âm $u_\alpha(t) = \|\mathbf{w}_t - \alpha\mathbf{w}^*\|_2^2$. Theo giả thiết phản chứng, tồn tại một điểm bị phân lớp lỗi khi dùng nghiệm \mathbf{w}_t . Giả sử đó là điểm \mathbf{x}_i với nhãn y_i . Ta có

$$u_\alpha(t+1) = \|\mathbf{w}_{t+1} - \alpha\mathbf{w}^*\|_2^2 \quad (13.9)$$

$$= \|\mathbf{w}_t + y_i\mathbf{x}_i - \alpha\mathbf{w}^*\|_2^2 \quad (13.10)$$

$$= \|\mathbf{w}_t - \alpha\mathbf{w}^*\|_2^2 + y_i^2\|\mathbf{x}_i\|_2^2 + 2y_i\mathbf{x}_i^T(\mathbf{w}_t - \alpha\mathbf{w}^*) \quad (13.11)$$

$$< u_\alpha(t) + \|\mathbf{x}_i\|_2^2 - 2\alpha y_i\mathbf{x}_i^T\mathbf{w}^* \quad (13.12)$$

Dấu nhỏ hơn ở dòng cuối là vì $y_i^2 = 1$ và $2y_i\mathbf{x}_i^T\mathbf{w}_t < 0$. Nếu tiếp tục đặt

$$\beta^2 = \max_{i=1,2,\dots,N} \|\mathbf{x}_i\|_2^2, \quad \gamma = \min_{i=1,2,\dots,N} y_i\mathbf{x}_i^T\mathbf{w}^* \quad (13.13)$$

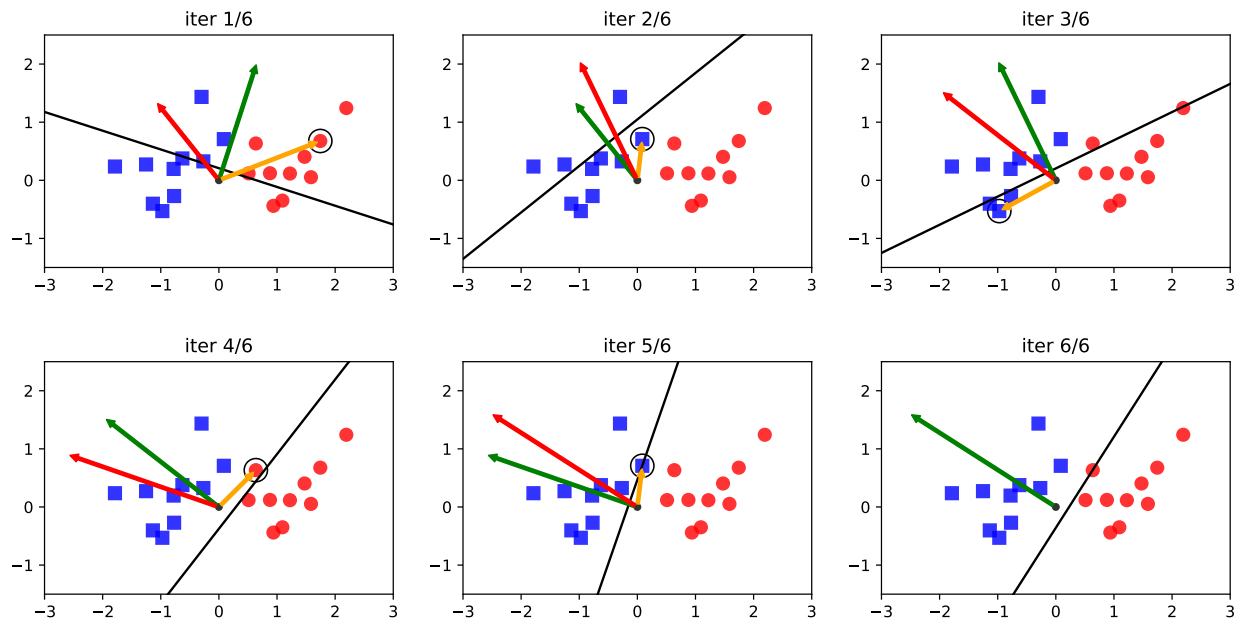
và chọn $\alpha = \frac{\beta^2}{\gamma}$, ta sẽ có $0 \leq u_\alpha(t+1) < u_\alpha(t) + \beta^2 - 2\alpha\gamma = u_\alpha(t) - \beta^2$. Ta có thể chọn giá trị này vì (13.12) đúng với α bất kỳ. Điều này chỉ ra rằng nếu luôn có điểm bị phân lớp lỗi thì dãy $u_\alpha(t)$ là một dãy giảm, bị chặn dưới bởi 0, và phần tử sau kém phần tử trước ít nhất một lượng là $\beta^2 > 0$. Điều vô lý này chứng tỏ đến một lúc nào đó sẽ không còn điểm nào bị phân lớp lỗi. Nói cách khác, thuật toán perceptron hội tụ sau một số hữu hạn bước.

13.3 Ví dụ và minh hoạ trên Python

Thuật toán 13.1 có thể được triển khai như sau:

```
import numpy as np
def predict(w, X):
    ''' predict label of each row of X, given w
    X: a 2-d numpy array of shape (N, d), each row is a datapoint
    w_init: a 1-d numpy array of shape (d) '''
    return np.sign(X.dot(w))

def perceptron(X, y, w_init):
    ''' perform perceptron learning algorithm
    X: a 2-d numpy array of shape (N, d), each row is a datapoint
    y: a 1-d numpy array of shape (N), label of each row of X. y[i] = 1/-1
    w_init: a 1-d numpy array of shape (d) '''
    w = w_init
    while True:
        pred = predict(w, X)
        # find indexes of misclassified points
        mis_idx = np.where(np.equal(pred, y) == False)[0]
        # number of misclassified points
        num_mis = mis_idx.shape[0]
        if num_mis == 0: # no more misclassified points
            return w
        # random pick one misclassified point
        random_id = np.random.choice(mis_idx, 1)[0]
        # update w
        w = w + y[random_id]*X[random_id]
```

Hình 13.3: Minh họa thuật toán perceptron. Các điểm màu lam thuộc lớp 1, các điểm màu đỏ thuộc lớp -1 . Tại mỗi vòng lặp, đường thẳng màu đen là đường ranh giới. Vector màu lục là w_t . Điểm được khoanh tròn là một điểm bị phân lớp lỗi x_i . Vector màu cam thể hiện vector x_i . Vector màu đỏ chính là w_{t+1} . Nếu $y_i = 1$ (màu lam), vector màu đỏ bằng tổng hai vector kia. Nếu $y_i = -1$, vector màu đỏ bằng hiệu hai vector kia.

Trong đó, hàm `predict(w, X)` dự đoán nhãn của mỗi hàng của X dựa trên công thức (13.2). Hàm `perceptron(X, y, w_init)` thực hiện thuật toán PLA với tập dữ liệu X , nhãn y và nghiệm ban đầu w_{init} .

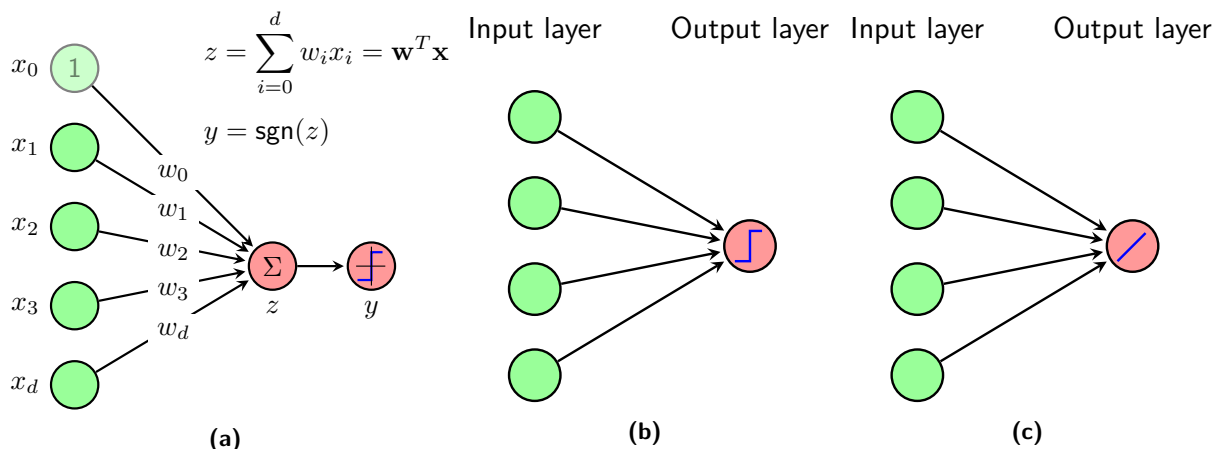
Để kiểm tra đoạn code trên, ta áp dụng nó vào một ví dụ với dữ liệu trong không gian hai chiều như dưới đây.

```
means = [[-1, 0], [1, 0]]
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)

X = np.concatenate((X0, X1), axis = 0)
y = np.concatenate((np.ones(N), -1*np.ones(N)))

Xbar = np.concatenate((np.ones((2*N, 1)), X), axis = 1)
w_init = np.random.randn(Xbar.shape[1])
w = perceptron(Xbar, y, w_init)
```

Mỗi lớp có 10 phần tử, là các vector ngẫu nhiên lấy theo phân phối chuẩn có ma trận hiệp phương sai `cov` và vector kỳ vọng được lưu trong `means`. Hình 13.3 minh họa nghiệm sau mỗi vòng lặp. Ta thấy rằng perceptron cho bài toán này hội tụ sau chỉ sau sáu vòng lặp.



Hình 13.4: Biểu diễn perceptron và linear regression dưới dạng neural network. (a) perceptron đầy đủ, (b) perceptron thu gọn, (c) linear regression thu gọn.

13.4 Mô hình neural network đầu tiên

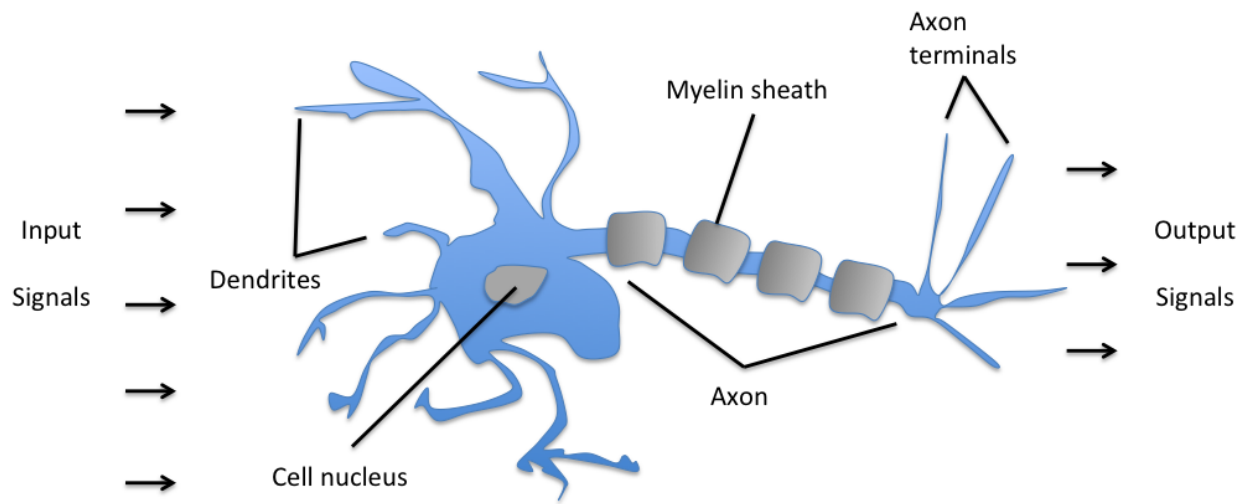
Hàm số dự đoán đầu ra của perceptron $\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x})$ có thể được mô tả trên Hình 13.4a. Đây chính là dạng đơn giản của neural network.

Đầu vào của network \mathbf{x} được minh họa bằng các *node* màu lục với node x_0 luôn luôn bằng 1. Tập hợp các node màu lục được gọi là *tầng đầu vào (input layer)*. Số node trong input layer là $d + 1$. Đôi khi node $x_0 = 1$ này được ẩn đi. Các *trọng số* w_0, w_1, \dots, w_d được gán vào các mũi tên đi tới node $z = \sum_{i=0}^d w_i x_i = \mathbf{w}^T \mathbf{x}$. Node $y = \text{sgn}(z)$ là *output* của network. Ký hiệu hình chữ Z ngược màu lam trong node y thể hiện đồ thị của hàm sgn . Hàm $y = \text{sgn}(z)$ đóng vai trò là một *hàm kích hoạt (activation function)*. Dữ liệu đầu vào được đặt vào input layer, lấy tổng có trọng số lưu vào biến z rồi đi qua hàm kích hoạt để có kết quả ở y . Đây chính là dạng đơn giản nhất của một neural network. Perceptron cũng có thể được vẽ giản lược như Hình 13.4b, với ẩn ý rằng dữ liệu ở input layer được lấy tổng có trọng số trước khi đi qua hàm lấy dấu $y = \text{sgn}(z)$.

Các neural network có thể có một hoặc nhiều node ở output tạo thành một *tầng đầu ra (output layer)*, hoặc có thể có thêm các layer trung gian giữa *input layer* và *output layer*, được gọi là *tầng ẩn (hidden layer)*. Các neural network thường có nhiều hidden layer và các layer có thể có các hàm kích hoạt khác nhau. Chúng ta sẽ đi sâu vào các neural network với nhiều hidden layer ở Chương 16. Trước đó, chúng ta sẽ tìm hiểu các neural network đơn giản hơn không có hidden layer.

Để ý rằng nếu ta thay *activation function* bởi hàm *identity* $y = z$, ta sẽ có một neural network mô tả linear regression như Hình 13.4c. Với đường thẳng chéo màu xanh thể hiện đồ thị hàm số $y = z$. Các trục tọa độ đã được lược bỏ.

Mô hình perceptron ở trên khá giống với một node nhỏ của dây thần kinh sinh học như Hình 13.5.



Schematic of a biological neuron.

Hình 13.5: Cấu trúc của một neuron thần kinh sinh học. Nguồn: *Single-Layer Neural Networks and Gradient Descent* (<https://goo.gl/RjBREb>).

Dữ liệu từ nhiều dây thần kinh (tương tự như x_i) đi về một *cell nucleus*. Cell nucleus đóng vai trò như bộ lấy tổng có trọng số $\sum_{i=0}^d w_i x_i$. Thông tin này sau đó được tổng hợp (tương tự như hàm kích hoạt) và đưa ra ở output. Tên gọi *neural network* hoặc *artificial neural network* được khởi nguồn từ đây.

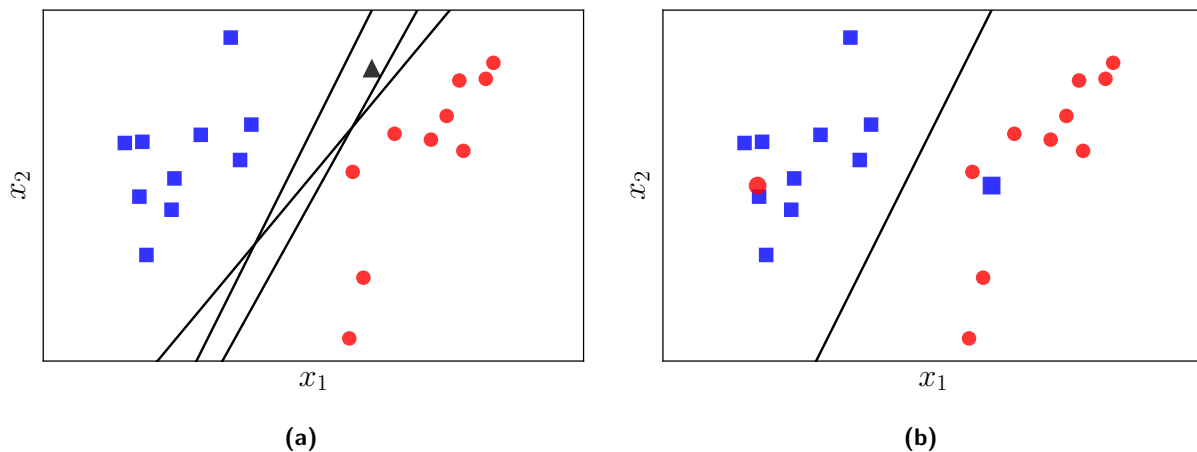
13.5 Thảo Luận

PLA có thể cho vô số nghiệm khác nhau. Nếu hai lớp dữ liệu là linearly separable thì có vô số đường thẳng ranh giới của hai lớp dữ liệu đó như trên Hình 13.6a. Tất cả các đường thẳng màu đen đều có thể đóng vai trò là đường ranh giới. Tuy nhiên, các đường khác nhau sẽ quyết định điểm hình tam giác thuộc các lớp khác nhau. Trong các đường đó, đường nào là tốt nhất? Và định nghĩa “tốt nhất” được hiểu theo nghĩa nào? Các câu hỏi này sẽ được thảo luận kỹ hơn trong Chương 26.

PLA đòi hỏi hai lớp dữ liệu phải linearly separable. Hình 13.6b mô tả hai lớp dữ liệu *tương đối* linearly separable. Mỗi lớp có một điểm coi như *nhiều* nằm lẫn trong các điểm của lớp kia. PLA sẽ không làm việc, tức không bao giờ dừng lại, trong trường hợp này vì với mọi đường thẳng ranh giới, luôn có ít nhất hai điểm bị phân lớp lỗi.

Trong một chừng mực nào đó, đường thẳng màu đen vẫn có thể coi là một nghiệm tốt vì nó đã giúp phân loại chính xác hầu hết các điểm. Việc không hội tụ với dữ liệu *gần* linearly separable chính là một nhược điểm lớn của PLA.

Nhược điểm này có thể được khắc phục bằng *pocket algorithm* dưới đây.



Hình 13.6: Với bài toán phân lớp nhị phân, PLA có thể (a) cho vô số nghiệm, hoặc (b) vô nghiệm thậm chí với chỉ một nhiễu nhỏ.

Pocket algorithm [AMMIL12]: một cách tự nhiên, nếu có một vài *nhiều*, ta sẽ đi tìm một đường thẳng phân chia hai class sao cho có ít điểm bị phân lớp lỗi nhất. Việc này có thể được thực hiện thông qua PLA với một chút thay đổi nhỏ:

1. Giới hạn số lượng vòng lặp của PLA. Đặt nghiệm \mathbf{w} sau vòng lặp đầu tiên và số điểm bị phân lớp lỗi vào trong *túi quần* (*pocket*).
2. Mỗi lần cập nhật nghiệm \mathbf{w}_t mới, ta đếm xem có bao nhiêu điểm bị phân lớp lỗi. So sánh số điểm bị phân lớp lỗi này với số điểm bị phân lớp lỗi trong *pocket*, nếu nhỏ hơn thì lấy nghiệm cũ ra, đặt nghiệm mới này vào. Lặp lại bước này đến khi hết số vòng lặp.

Thuật toán này giống với thuật toán tìm phần tử nhỏ nhất trong một mảng một chiều.

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/tisSTq>.

Logistic regression

14.1 Giới thiệu

14.1.1 Nhắc lại hai mô hình tuyến tính

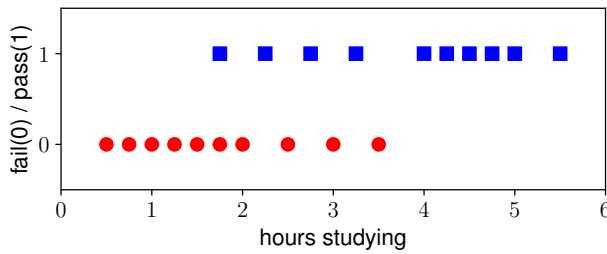
Hai mô hình tuyến tính đã thảo luận trong cuốn sách này, linear regression và perceptron (PLA), đều có thể viết chung dưới dạng $y = f(\mathbf{w}^T \mathbf{x})$ với $f(s)$ là một hàm kích hoạt. Với $f(s) = s$ trong linear regression, và $f(s) = \text{sgn}(s)$ trong PLA. Trong linear regression, tích vô hướng $\mathbf{w}^T \mathbf{x}$ được trực tiếp sử dụng để dự đoán output y , loại này phù hợp nếu ta cần dự đoán một đầu ra không bị chặn trên và dưới. Trong PLA, đầu ra chỉ nhận một trong hai giá trị 1 hoặc -1 , phù hợp với các bài toán phân lớp nhị phân. Trong chương này, chúng ta sẽ thảo luận một mô hình tuyến tính với một hàm kích hoạt khác, thường được áp dụng cho các bài toán phân lớp nhị phân. Trong mô hình này, đầu ra có thể được thể hiện dưới dạng xác suất. Ví dụ, xác suất thi đỗ nếu biết thời gian ôn thi, xác suất ngày mai có mưa dựa trên những thông tin đo được trong ngày hôm nay, v.v.. Mô hình này có tên là *logistic regression*. Mặc dù trong tên có chứa từ *regression*, logistic regression thường được sử dụng nhiều hơn cho các bài toán phân lớp.

14.1.2 Một ví dụ nhỏ

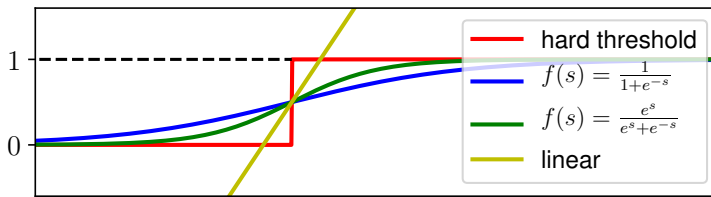
Bảng 14.1: Thời gian ôn thi (Hours) và kết quả thi của 20 sinh viên.

Hours	Pass	Hours	Pass	Hours	Pass	Hours	Pass
0.5	0	0.75	0	1	0	1.25	0
1.5	0	1.75	0	1.75	1	2	0
2.25	1	2.5	0	2.75	1	4	0
3.25	1	3.5	0	4	1	4.25	1
4.5	1	4.75	1	5	1	5.5	1

Xét một ví dụ về sự liên quan giữa thời gian ôn thi và kết quả thi của 20 sinh viên được cho trong Bảng 14.1. Bài toán đặt ra là từ dữ liệu này hãy xây dựng một mô hình đánh



Hình 14.1: Ví dụ về kết quả thi dựa trên số giờ ôn tập. Trục hoành thể hiện thời gian ôn tập của mỗi sinh viên, trục tung gồm hai giá trị 0/fail (các điểm màu đỏ) và 1/pass (các điểm màu xanh).



Hình 14.2: Một vài ví dụ về các hàm kích hoạt khác nhau.

giá khả năng đỗ của một sinh viên dựa trên thời gian ôn thi. Dữ liệu trong Bảng 14.1 được mô tả trên Hình 14.1. Nhìn chung, thời gian học càng nhiều thì khả năng đỗ càng cao. Tuy nhiên, không có một ngưỡng nào để có thể khẳng định rằng mọi sinh viên học nhiều thời gian hơn ngưỡng đó sẽ chắc chắn đỗ. Nói cách khác, dữ liệu của hai lớp này là không linearly separable, và vì vậy PLA sẽ không hữu ích ở đây. Tuy nhiên, thay vì dự đoán chính xác hai giá trị đỗ/trượt, ta có thể dự đoán xác suất để một sinh viên thi đỗ dựa trên thời gian ôn thi.

14.1.3 Mô hình logistic regression

Quan sát Hình 14.2 với các hàm kích hoạt $f(s)$ khác nhau.

- Đường màu vàng biểu diễn một hàm kích hoạt tuyến tính. Đường này không bị chặn nên không phù hợp cho bài toán đang xét với đầu ra là một giá trị trong khoảng $[0, 1]$.
- Đường màu đỏ tương tự với hàm kích hoạt của PLA với ngưỡng có thể khác không¹.
- Các đường màu lam và lục phù hợp với bài toán đang xét hơn. Chúng có một vài tính chất quan trọng:
 - Là các hàm số liên tục nhận giá trị thực, bị chặn trong khoảng $(0, 1)$.
 - Nếu coi điểm có tung độ là $1/2$ là ngưỡng, các điểm càng xa ngưỡng về phía bên trái có giá trị càng gần 0, các điểm càng xa ngưỡng về phía bên phải có giá trị càng gần 1. Điều này *khớp* với nhận xét rằng học càng nhiều thì xác suất đỗ càng cao và ngược lại.
 - Hai hàm này có đạo hàm mọi nơi, vì vậy có thể được lợi trong việc tối ưu.

¹ Đường này chỉ khác với activation function của PLA ở chỗ hai class là 0 và 1 thay vì -1 và 1.

Hàm sigmoid và tanh

Trong số các hàm số có ba tính chất nói trên, hàm *sigmoid*:

$$f(s) = \frac{1}{1 + e^{-s}} \triangleq \sigma(s) \quad (14.1)$$

được sử dụng nhiều nhất, vì nó bị chặn trong khoảng $(0, 1)$. Thêm nữa,

$$\lim_{s \rightarrow -\infty} \sigma(s) = 0; \quad \lim_{s \rightarrow +\infty} \sigma(s) = 1 \quad (14.2)$$

Thú vị hơn,

$$\sigma'(s) = \frac{e^{-s}}{(1 + e^{-s})^2} = \frac{1}{1 + e^{-s}} \frac{e^{-s}}{1 + e^{-s}} = \sigma(s)(1 - \sigma(s)) \quad (14.3)$$

Với đạo hàm đơn giản, hàm sigmoid được sử dụng rộng rãi trong neural network. Các bạn sẽ sớm thấy hàm sigmoid được khám phá ra như thế nào.

Ngoài ra, hàm *tanh* cũng hay được sử dụng: $\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$. Hàm số này nhận giá trị trong khoảng $(-1, 1)$. Bạn đọc có thể chứng minh được rằng $\tanh(s) = 2\sigma(2s) - 1$.

Hàm sigmoid có thể được thực hiện trên Python như sau.

```
def sigmoid(S):
    """
    S: an numpy array
    return sigmoid function of each element of S
    """
    return 1/(1 + np.exp(-S))
```

14.2 Hàm mất mát và phương pháp tối ưu

14.2.1 Xây dựng hàm mất mát

Với các mô hình với hàm mất mát màu lam và lục như trên, ta có thể giả sử rằng xác suất để một điểm dữ liệu \mathbf{x} rơi vào lớp thứ nhất là $f(\mathbf{w}^T \mathbf{x})$ và rơi vào lớp còn lại là $1 - f(\mathbf{w}^T \mathbf{x})$:

$$p(y_i = 1 | \mathbf{x}_i; \mathbf{w}) = f(\mathbf{w}^T \mathbf{x}_i) \quad (14.4)$$

$$p(y_i = 0 | \mathbf{x}_i; \mathbf{w}) = 1 - f(\mathbf{w}^T \mathbf{x}_i) \quad (14.5)$$

trong đó $p(y_i = 1 | \mathbf{x}_i; \mathbf{w})$ được hiểu là xác suất xảy ra sự kiện đầu ra $y_i = 1$ khi biết tham số mô hình \mathbf{w} và dữ liệu đầu vào \mathbf{x}_i . Mục đích cuối cùng là tìm các hệ số \mathbf{w} sao cho với các điểm dữ liệu ứng với $y_i = 1$, $f(\mathbf{w}^T \mathbf{x}_i)$ gần với 1, và ngược lại. Ký hiệu $z_i = f(\mathbf{w}^T \mathbf{x}_i)$, hai biểu thức (14.4) và (14.5) có thể được viết chung dưới dạng

$$p(y_i | \mathbf{x}_i; \mathbf{w}) = z_i^{y_i} (1 - z_i)^{1 - y_i} \quad (14.6)$$

Biểu thức này tương đương với hai biểu thức (14.4) và (14.5) ở trên vì khi $y_i = 1$, phần thứ hai của vế phải sẽ bằng 1, khi $y_i = 0$, phần thứ nhất sẽ bằng 1. Chúng ta muốn mô hình gần với dữ liệu đã cho nhất, tức xác suất này đạt giá trị cao nhất.

Xét toàn bộ test set với ma trận dữ liệu $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ và vector đầu ra tương ứng với mỗi cột $\mathbf{y} = [y_1, y_2, \dots, y_N]$. Ta cần giải bài toán tối ưu

$$\mathbf{w} = \arg \max_{\mathbf{w}} p(\mathbf{y}|\mathbf{X}; \mathbf{w}) \quad (14.7)$$

Đây chính là một bài toán maximum likelihood estimation với tham số mô hình \mathbf{w} cần được ước lượng. Giả sử rằng các điểm dữ liệu được sinh ra một cách ngẫu nhiên độc lập với nhau, ta có thể viết

$$p(\mathbf{y}|\mathbf{X}; \mathbf{w}) = \prod_{i=1}^N p(y_i|\mathbf{x}_i; \mathbf{w}) = \prod_{i=1}^N z_i^{y_i} (1 - z_i)^{1-y_i} \quad (14.8)$$

Lấy logarit tự nhiên, đổi dấu, và lấy trung bình, ta thu được hàm số

$$J(\mathbf{w}) = -\frac{1}{N} \log p(\mathbf{y}|\mathbf{X}; \mathbf{w}) = -\frac{1}{N} \sum_{i=1}^N (y_i \log z_i + (1 - y_i) \log(1 - z_i)) \quad (14.9)$$

với chú ý rằng z_i là một hàm số của \mathbf{w} và \mathbf{x}_i . Hàm số này chính là hàm mất mát của logistic regression. Ta cần đi tìm \mathbf{w} để $J(\mathbf{w})$ đạt giá trị nhỏ nhất (vì ta đã đổi dấu của biểu thức trong dấu argmax của (14.7)).

14.2.2 Tối ưu hàm mất mát

Bài toán tối ưu hàm mất mát của logistic regression có thể được giải quyết bằng stochastic gradient descent (SGD). Tại mỗi vòng lặp, \mathbf{w} sẽ được cập nhật dựa trên một điểm dữ liệu ngẫu nhiên. Hàm mất mát của logistic regression với chỉ một điểm dữ liệu (\mathbf{x}_i, y_i) và đạo hàm của nó lần lượt là

$$J(\mathbf{w}; \mathbf{x}_i, y_i) = -(y_i \log z_i + (1 - y_i) \log(1 - z_i)) \quad (14.10)$$

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i, y_i) = -\left(\frac{y_i}{z_i} - \frac{1 - y_i}{1 - z_i}\right) (\nabla_{\mathbf{w}} z_i) = \frac{z_i - y_i}{z_i(1 - z_i)} (\nabla_{\mathbf{w}} z_i) \quad (14.11)$$

ở đây ta đã sử dụng quy tắc chuỗi để tính đạo hàm với $z_i = f(\mathbf{w}^T \mathbf{x})$. Để cho biểu thức này trở nên gọn và đẹp hơn, ta sẽ tìm hàm $z = f(\mathbf{w}^T \mathbf{x})$ sao cho mẫu số bị triệt tiêu.

Nếu đặt $s = \mathbf{w}^T \mathbf{x}$, ta sẽ có

$$\nabla_{\mathbf{w}} z_i = \frac{\partial z_i}{\partial s} (\nabla_{\mathbf{w}} s) = \frac{\partial z_i}{\partial s} \mathbf{x}_i \quad (14.12)$$

Một cách tự nhiên, ta sẽ tìm hàm số $z = f(s)$ sao cho:

$$\frac{\partial z}{\partial s} = z(1 - z) \quad (14.13)$$

để triệt tiêu mẫu số trong biểu thức (14.11). Phương trình vi phân này không quá phức tạp. Thật vậy, (14.13) tương đương với

$$\begin{aligned}
& \frac{\partial z}{z(1-z)} = \partial s \\
\Leftrightarrow & \left(\frac{1}{z} + \frac{1}{1-z}\right)\partial z = \partial s \\
\Leftrightarrow & \log z - \log(1-z) = s + C \\
\Leftrightarrow & \log \frac{z}{1-z} = s + C \\
\Leftrightarrow & \frac{z}{1-z} = e^{s+C} \\
\Leftrightarrow & z = e^{s+C}(1-z) \\
\Leftrightarrow & z = \frac{e^{s+C}}{1+e^{s+C}} = \frac{1}{1+e^{-s-C}} = \sigma(s+C)
\end{aligned}$$

với C là một hằng số. Đơn giản chọn $C = 0$, ta được $z = f(\mathbf{w}^T \mathbf{x}) = \sigma(s)$. Đây chính là lý do hàm sigmoid được ra đời. Logistic regression với hàm kích hoạt là hàm sigmoid được sử dụng phổ biến nhất. Mô hình này còn có tên là *logistic sigmoid regression*. Khi nói logistic regression, ta ngầm hiểu rằng đó chính là logistic sigmoid regression.

Thay (14.12) và (14.13) vào (14.11) ta thu được

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i, y_i) = (z_i - y_i)\mathbf{x}_i = (\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i \quad (14.14)$$

Và công thức cập nhật nghiệm cho logistic sigmoid regression sử dụng SGD là

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(z_i - y_i)\mathbf{x}_i = \mathbf{w} - \eta(\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i \quad (14.15)$$

với η là một learning rate dương.

14.2.3 Logistic regression với weight decay

Một trong các kỹ thuật phổ biến giúp tránh overfitting với các neural network là sử dụng *weight decay*. Weight decay là một kỹ thuật regularization, trong đó một đại lượng tỉ lệ với bình phương norm 2 của vector hệ số được cộng vào hàm mất mát để hạn chế độ lớn của các hệ số. Hàm mất mát trở thành

$$\bar{J}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \left(-y_i \log z_i - (1 - y_i) \log(1 - z_i) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right) \quad (14.16)$$

Công thức cập nhật SGD cho \mathbf{w} với hàm này cũng đơn giản vì phần regularization có đạo hàm đơn giản:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left((\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i + \lambda \mathbf{w} \right) \quad (14.17)$$

14.3 Triển khai thuật toán trên Python

Hàm ước lượng xác suất cho mỗi điểm dữ liệu và hàm tính giá trị hàm mất mát với weight decay có thể được thực hiện như sau trong Python.

```
def prob(w, X):
    """
    X: a 2d numpy array of shape (N, d). N datapoint, each with size d
    w: a 1d numpy array of shape (d)
    """
    return sigmoid(X.dot(w))

def loss(w, X, y, lam):
    """
    X, w as in prob
    y: a 1d numpy array of shape (N). Each elem = 0 or 1
    """
    z = prob(w, X)
    return -np.mean(y*np.log(z) + (1-y)*np.log(1-z)) + 0.5*lam/X.shape[0]*np.sum(w*w)
```

Từ công thức (14.17), ta có thể thực hiện thuật toán tìm w cho logistic regression như sau.

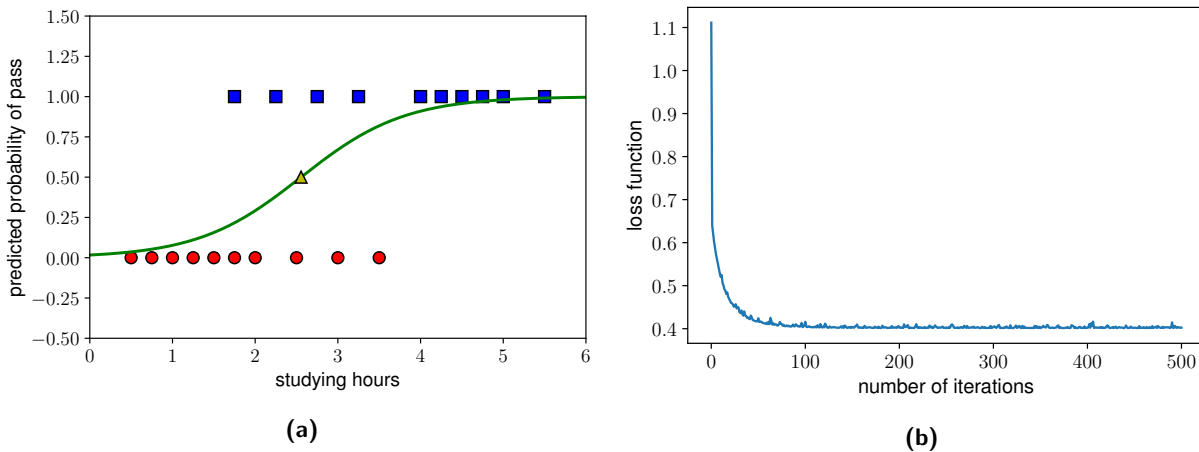
```
def logistic_regression(w_init, X, y, lam = 0.001, lr = 0.1, nepoches = 2000):
    # lam - reg parameter, lr - learning rate, nepoches - number of epochs
    N, d = X.shape[0], X.shape[1]
    w = w_old = w_init
    loss_hist = [loss(w_init, X, y, lam)] # store history of loss in loss_hist
    ep = 0
    while ep < nepoches:
        ep += 1
        mix_ids = np.random.permutation(N)
        for i in mix_ids:
            xi = X[i]
            yi = y[i]
            zi = sigmoid(xi.dot(w))
            w = w - lr*((zi - yi)*xi + lam*w)
        loss_hist.append(loss(w, X, y, lam))
        if np.linalg.norm(w - w_old)/d < 1e-6:
            break
        w_old = w
    return w, loss_hist
```

14.3.1 Logistic regression cho ví dụ ban đầu

Áp dụng vào bài toán ví dụ ban đầu.

```
X = np.array([[0.50, 0.75, 1.00, 1.25, 1.50, 1.75, 1.75, 2.00, 2.25, 2.50,
                2.75, 3.00, 3.25, 3.50, 4.00, 4.25, 4.50, 4.75, 5.00, 5.50]]).T
y = np.array([0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1])

# bias trick
Xbar = np.concatenate((X, np.ones((N, 1))), axis = 1)
w_init = np.random.randn(Xbar.shape[1])
lam = 0.0001
w, loss_hist = logistic_regression(w_init, Xbar, y, lam, lr = 0.05, nepoches = 500)
print(w)
print(loss(w, Xbar, y, lam))
```



Hình 14.3: Nghiệm của logistic regression cho bài toán dự đoán kết quả thi dựa trên thời gian học. (a) Đường màu lục thể hiện xác suất thi đỗ dựa trên thời gian học. Điểm tam giác vàng thể hiện ngưỡng ra quyết định đỗ/trượt. Điểm này có thể thay đổi tùy vào bài toán. (b) Giá trị của hàm mất mát qua các vòng lặp. Hàm mất mát giảm rất nhanh và hội tụ rất sớm.

Kết quả:

```
Solution of Logistic Regression: [ 1.54337021 -4.06486702]
Final loss: 0.402446724975
```

Từ đây ta có thể rút ra xác suất thi đỗ dựa trên công thức:

$$\text{probability_of_pass} \approx \text{sigmoid}(1.54 * \text{hours_of_studying} - 4.06)$$

Biểu thức này cũng chỉ ra rằng xác suất thi đỗ tăng khi thời gian ôn tập tăng, do sigmoid là một hàm đồng biến. Nghiệm của mô hình logistic regression và giá trị hàm mất mát qua mỗi epoch được mô tả trên Hình 14.3.

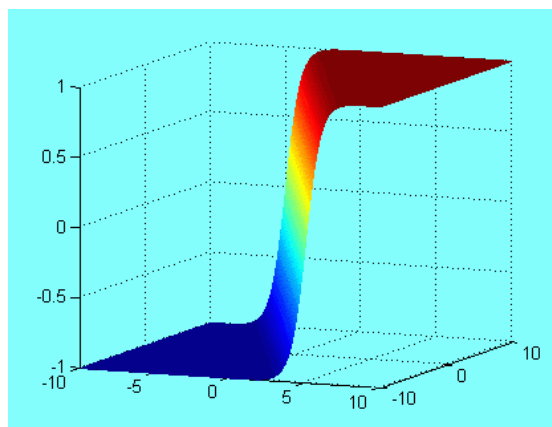
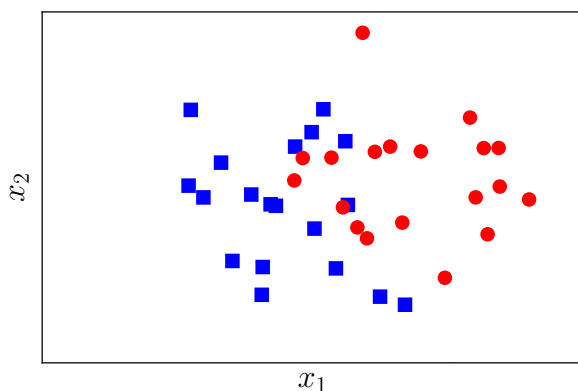
14.3.2 Ví dụ với dữ liệu hai chiều

Chúng ta xét thêm một ví dụ nhỏ trong không gian hai chiều. Giả sử có hai lớp xanh và đỏ với dữ liệu được phân bố như Hình 14.4a. Với dữ liệu đầu vào nằm trong không gian hai chiều, hàm sigmoid có dạng như thác nước như Hình 14.4b.

Kết quả tìm được khi áp dụng mô hình logistic regression được minh họa như Hình 14.5 với màu nền thể hiện xác suất điểm đó thuộc class đỏ. Màu xanh đậm thể hiện giá trị 0, màu đỏ đậm thể hiện giá trị rất gần với 1.

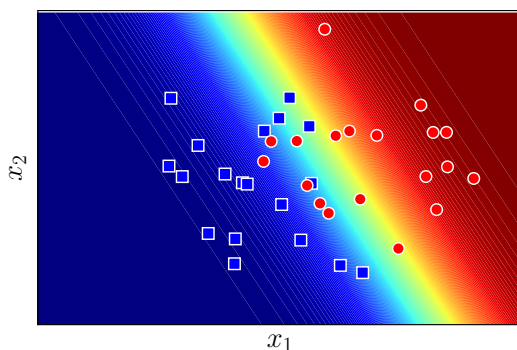
Khi phải lựa chọn một ranh giới thay vì xác suất, ta quan sát thấy các đường thẳng nằm trong khu vực lục và vàng là một lựa chọn hợp lý. Ta sẽ chứng minh ở phần sau rằng tập hợp các điểm cho cùng xác suất đầu ra tạo thành một siêu phẳng.

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/9e7sPF>.



(a) Dữ liệu cho bài toán phân lớp trong không gian hai chiều. (b) Đồ thị hàm sigmoid trong không gian hai chiều.

Hình 14.4: Ví dụ về dữ liệu trong không gian hai chiều và hàm sigmoid trong không gian đó.



Hình 14.5: Ví dụ về Logistic Regression với dữ liệu hai chiều. Vùng màu đỏ càng đậm thể hiện xác suất thuộc lớp dữ liệu đỏ càng cao, vùng màu xanh càng đậm thể hiện xác suất thuộc lớp dữ liệu đỏ càng thấp - tức xác suất thuộc lớp dữ liệu xanh càng cao. Vùng biên giữa hai lớp thể hiện các điểm thuộc vào mỗi lớp với xác suất thấp hơn (độ tin cậy thấp hơn).

Cách sử dụng logistic regression trong thư viện scikit-learn có thể được tìm thấy tại <https://goo.gl/BJLJNx>.

14.4 Tính chất của logistic regression

1. Logistic Regression được sử dụng nhiều trong các bài toán Classification.

Mặc dù trong tên có từ regression, logistic regression được sử dụng nhiều trong các bài toán classification. Sau khi tìm được mô hình, việc xác định class y cho một điểm dữ liệu \mathbf{x} được xác định bằng việc so sánh hai biểu thức xác suất

$$P(y = 1|\mathbf{x}; \mathbf{w}); \quad P(y = 0|\mathbf{x}; \mathbf{w}) \quad (14.18)$$

Nếu biểu thức thứ nhất lớn hơn, ta kết luận điểm dữ liệu thuộc class 1, và ngược lại. Vì tổng hai biểu thức này luôn bằng một nên một cách gọn hơn, ta chỉ cần xác định xem $P(y = 1|\mathbf{x}; \mathbf{w})$ lớn hơn 0.5 hay không.

2. Đường ranh giới tạo bởi logistic regression là một siêu phẳng

Thật vậy, giả sử những điểm có xác suất đầu ra lớn hơn 0.5 được coi là thuộc vào lớp có nhãn là 1. Tập hợp các điểm này là nghiệm của bất phương trình

$$P(y = 1|\mathbf{x}; \mathbf{w}) > 0.5 \Leftrightarrow \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} > 0.5 \Leftrightarrow e^{-\mathbf{w}^T \mathbf{x}} < 1 \Leftrightarrow \mathbf{w}^T \mathbf{x} > 0 \quad (14.19)$$

Nói cách khác, tập hợp các điểm thuộc lớp 1 tạo thành *nửa không gian (halfspace)* $\mathbf{w}^T \mathbf{x} > 0$, tập hợp các điểm thuộc lớp 0 tạo thành nửa không gian còn lại. Ranh giới giữa hai lớp chính là siêu phẳng $\mathbf{w}^T \mathbf{x} = 0$.

Chính vì điều này, logistic regression được coi như một bộ phân lớp tuyến tính.

3. Logistic regression không yêu cầu giả thiết linearly separable.

Một điểm cộng của logistic regression so với PLA là nó không cần có giả thiết dữ liệu hai lớp là linearly separable. Tuy nhiên, ranh giới tìm được vẫn có dạng tuyến tính. Vì vậy, mô hình này chỉ phù hợp với loại dữ liệu mà hai lớp gần với linearly separable, tức chỉ có một vài điểm dữ liệu phá vỡ tính linearly separable của hai lớp.

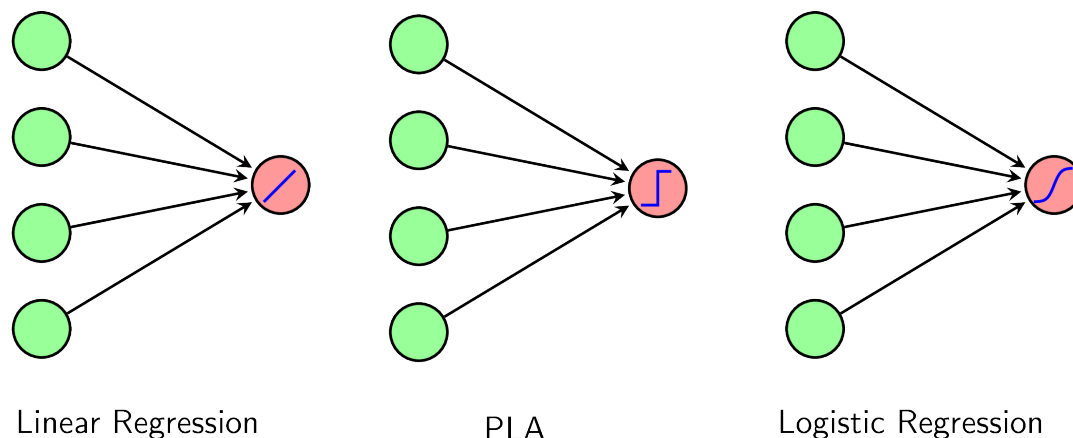
4. Ngưỡng ra quyết định có thể thay đổi.

Hàm dự đoán đầu ra của các điểm dữ liệu mới có thể được viết như sau.

```
def predict(w, X, threshold = 0.5):
    """
    predict output of each row of X
    X: a numpy array of shape (N, d)
    threshold: a threshold between 0 and 1
    return a 1d numpy array, each element is 0 or 1
    """
    res = np.zeros(X.shape[0])
    res[np.where(prob(w, X) > threshold)[0]] = 1
    return res
```

Trong các ví dụ đã nêu, ngưỡng ra quyết định đều được lấy tại 0.5. Trong nhiều trường hợp, ngưỡng này có thể được thay đổi. Ví dụ, việc xác định các giao dịch là lừa đảo của một công ty tín dụng là rất quan trọng. Việc phân lớp nhầm một giao dịch lừa đảo thành một giao dịch thông thường gây ra hậu quả nghiêm trọng hơn chiều ngược lại. Trong bài toán đó, ngưỡng phân loại có thể giảm xuống còn 0.3. Nghĩa là các giao dịch được dự đoán là lừa đảo với xác suất lớn hơn 0.3 sẽ được xếp vào lớp lừa đảo và nên được tiếp tục đánh giá bằng các bước khác.

5. Khi biểu diễn theo neural networks, linear regression, PLA, và logistic regression có thể được biểu diễn như trên Hình 14.6. Sự khác nhau chỉ nằm ở lựa chọn hàm kích hoạt.



Hình 14.6: Biểu diễn linear regression, PLA, và logistic regression dưới dạng neural network.

14.5 Bài toán phân biệt hai chữ số viết tay

Chúng ta cùng làm một ví dụ thực tế hơn với bài toán phân biệt hai chữ số 0 và 1 trong bộ cơ sở dữ liệu MNIST. Trong mục này, chúng ta sẽ sử dụng class **LogisticRegression** trong thư viện scikit-learn. Trước tiên, ta khai báo các thư viện và tải về bộ cơ sở dữ liệu MNIST.

```
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
mnist = fetch_mldata('MNIST original', data_home='../data/')
N, d = mnist.data.shape
```

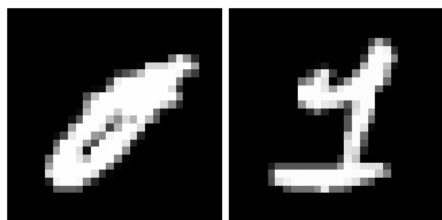
Có tổng cộng 70000 điểm dữ liệu trong tập dữ liệu MNIST, mỗi điểm là một mảng 784 phần tử tương ứng với 784 pixel. Mỗi chữ số từ 0 đến 9 chiếm khoảng 10%. Chúng ta sẽ lấy ra tất cả các điểm ứng với chữ số 0 và 1, sau đó lấy ra ngẫu nhiên 2000 điểm làm test set, phần còn lại đóng vai trò training set.

```
X_all = mnist.data
y_all = mnist.target

X0 = X_all[np.where(y_all == 0)[0]] # all digit 0
X1 = X_all[np.where(y_all == 1)[0]] # all digit 1
y0 = np.zeros(X0.shape[0]) # class 0 label
y1 = np.ones(X1.shape[0]) # class 1 label

X = np.concatenate((X0, X1), axis = 0) # all digits
y = np.concatenate((y0, y1)) # all labels

# split train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=2000)
```



Hình 14.7: Các chữ số bị phân lớp lỗi trong bài toán phân lớp nhị phân với hai chữ số 0 và 1.

Tiếp theo, ta xây dựng mô hình logistic regression trên test set và dự đoán nhãn của các điểm trong test set. Kết quả này được so sánh với nhãn thật của mỗi điểm dữ liệu để tính độ chính xác của bộ phân lớp trên tập kiểm thử.

```
model = LogisticRegression(C = 1e5) # C is inverse of lam
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("Accuracy %.2f %" % (100*accuracy_score(y_test, y_pred.tolist())))
```

Tuyệt vời, gần như 100% được phân loại chính xác. Điều này là dễ hiểu vì hai chữ số 0 và 1 khác nhau rất nhiều.

Tiếp theo, ta cùng đi tìm những ảnh bị phân loại sai và hiển thị chúng.

```
mis = np.where((y_pred - y_test) != 0)[0]
Xmis = X_test[mis, :]

from display_network import *
filename = 'mnist_mis.pdf'
with PdfPages(filename) as pdf:
    plt.axis('off')
    A = display_network(Xmis.T, 1, Xmis.shape[0])
    f2 = plt.imshow(A, interpolation='nearest')
    plt.gray()
    pdf.savefig(bbox_inches='tight')
    plt.show()
```

Chỉ có hai chữ số bị phân lớp lỗi được cho trên Hình 14.7. Trong đó, chữ số 0 bị phân lớp lỗi là dễ hiểu vì nó trông rất giống chữ số 1.

Bạn đọc có thể xem thêm ví dụ về bài toán xác định giới tính dựa trên ảnh khuôn mặt tại <https://goo.gl/9V8wdD>.

14.6 Bộ phân lớp nhị phân cho bài toán phân lớp đa lớp

Logistic regression được áp dụng cho các bài toán phân lớp nhị phân. Các bài toán phân lớp thực tế có thể có nhiều hơn hai lớp dữ liệu rất nhiều, được gọi là bài toán *phân lớp đa lớp*

(*multi-class classification*). Với một vài kỹ thuật nhỏ, ta có thể áp dụng logistic regression cho các bài toán phân lớp đa lớp.

Có *ít nhất* bốn cách để áp dụng logistic regression hay các bộ phân lớp nhị phân vào các bài toán phân lớp đa lớp.

14.6.1 One-vs-one

Xây dựng rất nhiều các bộ phân lớp nhị phân cho từng cặp hai lớp dữ liệu. Bộ thứ nhất phân biệt lớp thứ nhất và thứ hai, bộ thứ hai phân biệt lớp thứ nhất và lớp thứ ba, v.v.. Dữ liệu mới được đưa vào tất cả các bộ phân lớp nhị phân nêu trên. Kết quả cuối cùng có thể được xác định bằng cách xem lớp nào mà điểm dữ liệu đó được phân vào nhiều nhất. Hoặc với logistic regression thì ta có thể tính *tổng các xác suất* mà điểm dữ liệu đó rơi vào mỗi lớp. Như vậy, nếu có C lớp thì số bộ phân lớp cần dùng là $\frac{C(C-1)}{2}$. Đây là một con số lớn, cách làm này không lợi về tính toán. Hơn nữa, nếu một chữ số thực ra là chữ số **1**, nhưng lại được đưa vào bộ phân lớp giữa các chữ số **5** và **6**, thì cả hai khả năng đều không hợp lý.

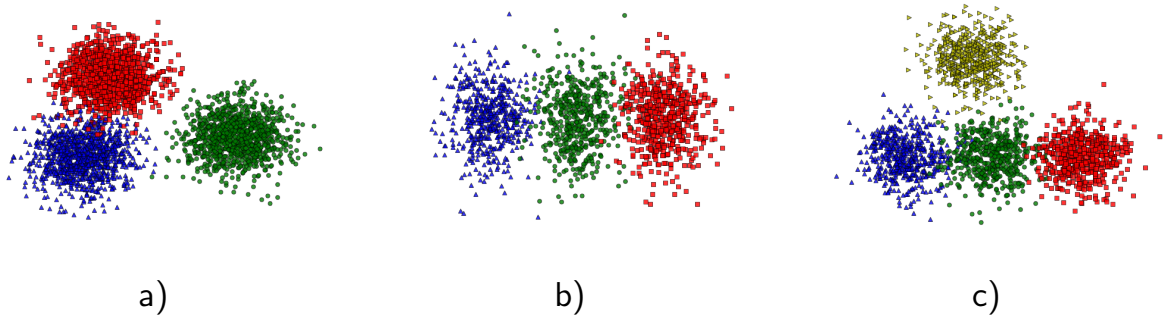
14.6.2 Phân tầng

Cách làm như **one-vs-one** sẽ mất rất nhiều thời gian huấn luyện vì có quá nhiều bộ phân lớp cần được xây dựng. Một cách giúp giảm số bộ phân lớp nhị phân là *phân tầng (hierarchical)*. Ý tưởng của phương pháp này có thể được thấy qua ví dụ sau.

Giả sử ta có bài toán phân lớp 4 chữ số 4, 5, 6, 7 trong MNIST. Vì chữ số 4 và 7 khá giống nhau, chữ số 5 và 6 khá giống nhau nên trước tiên ta xây dựng bộ phân lớp [4, 7] vs [5, 6]. Sau đó xây dựng thêm hai bộ 4 vs 7 và 5 vs 6. Tổng cộng, ta cần ba bộ phân lớp. Chú ý rằng có nhiều cách chia khác nhau, ví dụ [4, 5, 6] vs 7, [4, 5] vs 6, rồi 4 vs 5. Ưu điểm của kỹ thuật này là nó sử dụng ít bộ phân lớp tuyến tính hơn *one-vs-one*. Hạn chế lớn nhất của nó là việc nếu chỉ một bộ phân lớp cho kết quả sai thì kết quả cuối cùng chắc chắn sẽ sai. Ví dụ, nếu một ảnh chứa chữ số 5 bị phân lớp lỗi bởi bộ phân lớp đầu tiên, nó sẽ bị phân sang nhánh [4, 7]. Cuối cùng, nó sẽ bị phân vào lớp 4 hoặc 7, cả hai đều không chính xác.

14.6.3 Binary coding

Có một cách giảm số binary classifiers hơn nữa là *binary coding*, tức *mã hóa* output của mỗi lớp bằng một số nhị phân. Ví dụ, nếu có bốn lớp dữ liệu thì các lớp được mã hoá là 00, 01, 10, và 11. Với cách làm này, số bộ phân lớp nhị phân cần xây dựng chỉ là $m = \lceil \log_2(C) \rceil$ trong đó C là số lớp, $\lceil a \rceil$ là *số nguyên nhỏ nhất không nhỏ hơn a*. Bộ thứ nhất đi tìm bit đầu tiên của output (đã được mã hóa nhị phân), bộ thứ hai sẽ đi tìm bit thứ hai, v.v.. Cách làm này sử dụng một số lượng nhỏ nhất các bộ phân lớp tuyến tính. Tuy nhiên, nó có một hạn chế rất lớn là chỉ cần một bit bị phân loại sai sẽ dẫn đến dữ liệu bị phân loại sai. Hơn nữa, nếu số lớp không phải là lũy thừa của hai, mã nhị phân nhận được có thể là một giá trị không tương ứng với lớp nào.



Hình 14.8: Một số ví dụ về phân phối của các lớp dữ liệu trong bài toán phân lớp đa lớp.

14.6.4 one-vs-rest hay one-hot coding

Kỹ thuật được sử dụng nhiều nhất là **one-vs-rest** (một số tài liệu gọi là **one-vs-all**, **one-against-rest**, hoặc **one-against-all**). Cụ thể, nếu có C lớp thì ta sẽ xây dựng C bộ phân lớp nhị phân, mỗi bộ tương ứng với một lớp. Bộ thứ nhất giúp phân biệt lớp thứ nhất với các lớp còn lại, tức xem một điểm có thuộc lớp thứ nhất hay không, hoặc xác suất để một điểm rơi vào lớp đó là bao nhiêu. Tương tự như thế, bộ thứ hai sẽ phân biệt lớp thứ hai với các lớp còn lại, v.v.. Kết quả cuối cùng có thể được xác định bằng cách xác định lớp mà một điểm rơi vào với xác suất cao nhất.

Logistic regression trong thư viện sklearn có thể được dùng trực tiếp để áp dụng vào các bài toán phân lớp đa lớp với kỹ thuật **one-vs-rest**. Với bài toán MNIST, để dùng logistic regression kết hợp với one-vs-rest (mặc định trong scikit-learn), ta có thể làm như sau.

```
# all class
X_train, X_test, y_train, y_test = train_test_split(X_all, y_all, test_size=10000)
model = LogisticRegression(C = 1e5) # C is inverse of lam
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy %.2f %" % (100*accuracy_score(y_test, y_pred.tolist())))
```

Kết quả thu được khoảng 91.7%. Đây vẫn là một kết quả quá thấp so với con số 99.7% mà deep learning đã đạt được. Ngay cả K-nearest neighbors cũng đã đạt khoảng 96 %.

14.7 Thảo luận

14.7.1 Kết hợp các phương pháp trên

Trong nhiều trường hợp, ta cần phải kết hợp hai hoặc ba trong số bốn kỹ thuật đã đề cập. Xét ba ví dụ trong Hình 14.8.

- Hình 14.8a: cả 4 phương pháp trên đây đều có thể áp dụng được.

- Hình 14.8b: one-vs-rest không phù hợp vì lớp màu lục và hợp của lớp lam và lớp đỏ là không (gần) *linearly separable*. Lúc này, one-vs-one hoặc hierarchical phù hợp hơn.
- Hình 14.8c: Tương tự như trên, ba lớp lam, lục, đỏ thẳng hàng nên sẽ không dùng được one-vs-rest. Trong khi đó, one-vs-one vẫn hiệu quả vì từng cặp lớp dữ liệu là *linearly separable*. Tương tự hierarchical cũng làm việc nếu ta phân chia các nhóm một cách hợp lý. Hoặc chúng ta có thể kết hợp nhiều phương pháp. Ví dụ: dùng one-vs-rest để tìm *đỏ* với *không đỏ*. Nếu một điểm dữ liệu là *không đỏ*, với ba lớp còn lại, ta lại quay lại trường hợp Hình 14.8a và có thể dùng các phương pháp khác. Nhưng khó khăn vẫn nằm ở việc phân nhóm như thế nào, liệu rằng những lớp nào có thể cho vào cùng một nhóm?

Với bài toán phân lớp đa lớp, nhìn chung các kỹ thuật sử dụng các bộ phân lớp nhị phân đã trở nên ít hiệu quả hơn so với các phương pháp mới. Mời bạn đọc thêm Chương 15 và Chương 29 để tìm hiểu về các bộ phân lớp đa lớp phổ biến nhất hiện nay.

14.7.2 Biểu diễn các kỹ thuật đã nêu dưới dạng neural network

Lấy ví dụ với bài toán có bốn lớp dữ liệu 1, 2, 3, 4; ta có thể biểu diễn các mô hình được đề cập trong Mục 14.6 dưới dạng neural network như trên Hình 14.9. Các node màu đỏ thể hiện đầu ra là một trong hai giá trị.

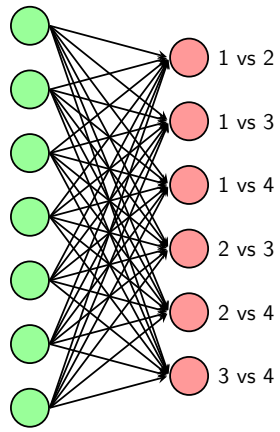
Các network này đều có nhiều node ở output layer, và một vector trọng số \mathbf{w} bây giờ đã trở thành *ma trận trọng số* \mathbf{W} mà mỗi cột của nó tương ứng với vector trọng số của một node output. Việc tối ưu đồng thời các bộ phân lớp nhị phân trong mỗi network cũng được tổng quát lên nhờ các phép tính với ma trận. Lúc này, công thức cập nhật của logistic regression

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(z_i - y_i)\mathbf{x}_i \quad (14.20)$$

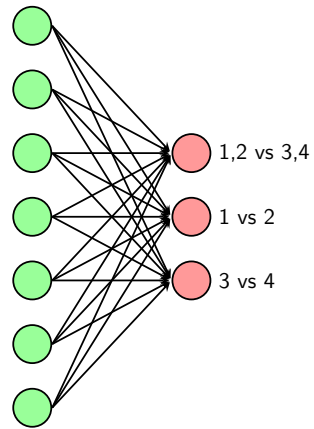
có thể được tổng quát thành

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{x}_i (\mathbf{z}_i - \mathbf{y}_i)^T \quad (14.21)$$

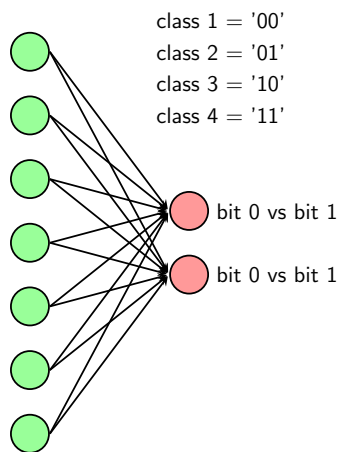
Với \mathbf{W} , \mathbf{y}_i , \mathbf{z}_i lần lượt là ma trận trọng số, vector (cột) output *thật* với toàn bộ các bộ phân lớp nhị phân tương ứng với điểm dữ liệu \mathbf{x}_i , và vector output tìm được của networks tại thời điểm đang xét nếu đầu vào mỗi network là \mathbf{x}_i . Chú ý rằng vector \mathbf{y}_i là một vector nhị phân, vector \mathbf{z}_i gồm các phần tử nằm trong khoảng $(0, 1)$.



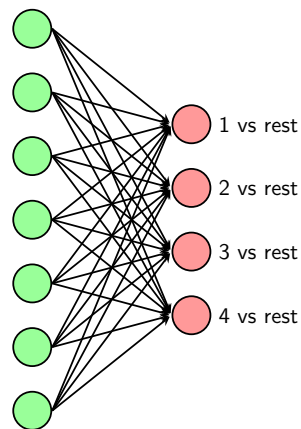
one-vs-one



Hierarchical



Binary coding



one-vs-rest

Hình 14.9: Mô hình neural networks cho các kỹ thuật sử dụng các bộ phân lớp nhị phân cho bài toán phân lớp đa lớp.

Softmax regression

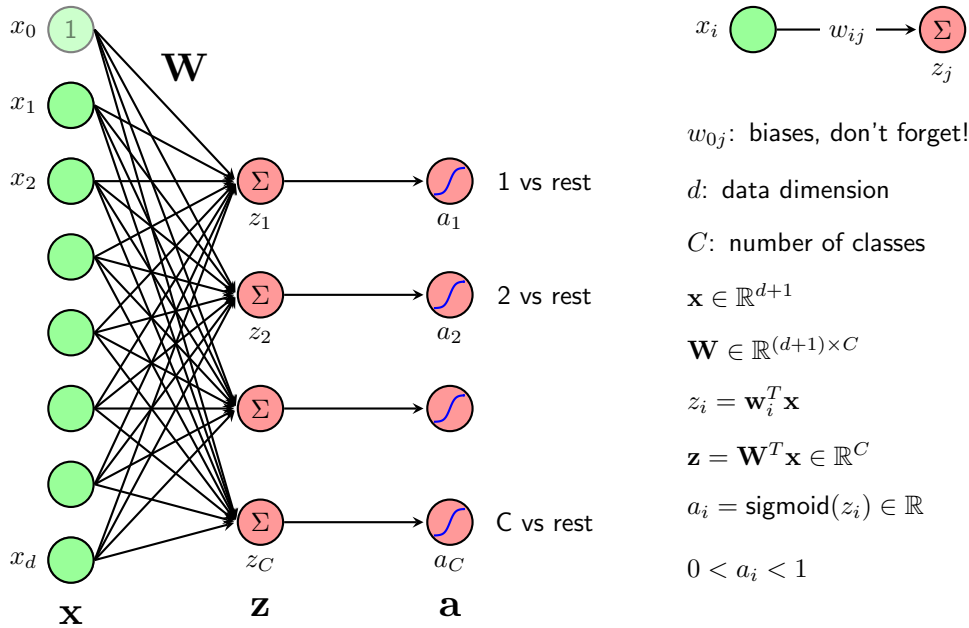
Các bài toán phân lớp thực tế thường có rất nhiều lớp dữ liệu. Như đã thảo luận trong Chương 14, các bộ phân lớp nhị phân tuy có thể kết hợp được với nhau để giải quyết các bài toán này, chúng vẫn có những hạn chế nhất định. Trong chương này, một phương pháp mở rộng của logistic regression có tên là *softmax regression* sẽ được giới thiệu nhằm khắc phục những hạn chế đã đề cập. Một lần nữa, mặc dù trong tên có chứa từ “regression”, softmax regression được sử dụng cho các bài toán phân lớp. Nó cũng chính là một trong những thành phần phổ biến nhất trong các bộ phân lớp hiện nay.

15.1 Giới thiệu

Với bài toán phân lớp nhị phân sử dụng logistic regression, đầu ra của neural network là một số thực trong khoảng $(0, 1)$, đóng vai trò như là xác suất để đầu vào thuộc một trong hai lớp. Ý tưởng này cũng có thể mở rộng cho bài toán phân lớp đa lớp, ở đó có C node ở output layer và giá trị mỗi node đóng vai trò như xác suất để đầu vào rơi vào lớp tương ứng. Như vậy, các đầu ra này liên kết với nhau qua việc chúng đều là các số dương và có tổng bằng một. Mô hình softmax regression thảo luận trong chương này đảm bảo tính chất đó.

Nhắc lại kỹ thuật *one-vs-rest* được trình bày trong chương trước được biểu diễn dưới dạng neuron network như trong Hình 15.1. Output layer màu đỏ có thể phân tách thành hai *sublayer* và mỗi thành phần của sublayer thứ hai a_i chỉ phụ thuộc vào thành phần tương ứng ở sublayer thứ nhất z_i thông qua hàm sigmoid $a_i = \sigma(z_i)$. Các giá trị đầu ra a_i đều là các số dương nhưng vì không có ràng buộc giữa chúng, tổng của chúng có thể là một số dương bất kỳ.

Chú ý rằng các mô hình linear regression, PLA, và logistic regression chỉ có một node ở output layer. Trong các trường hợp đó, tham số mô hình chỉ là một vector \mathbf{w} . Trong trường hợp output layer có nhiều hơn một node, tham số mô hình sẽ là tập hợp tham số \mathbf{w}_i ứng với từng node. Lúc này, ta có một *ma trận trọng số* (*weight matrix*) $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_C]$, mỗi cột ứng với một node ở output layer.



Hình 15.1: Phân lớp đa lớp với logistic regression và one-vs-rest.

15.2 Softmax function

15.2.1 Công thức của Softmax function

Chúng ta cần một mô hình xác suất sao cho với mỗi input \mathbf{x} , a_i thể hiện xác suất để input đó rơi vào lớp thứ i . Vậy điều kiện cần là các a_i phải dương và tổng của chúng bằng một. Ngoài ra, ta sẽ thêm một điều kiện cũng rất tự nhiên nữa, đó là giá trị $z_i = \mathbf{w}_i^T \mathbf{x}$ càng lớn thì xác suất dữ liệu rơi vào lớp thứ i càng cao. Điều kiện cuối này chỉ ra rằng ta cần một quan hệ đồng biến ở đây.

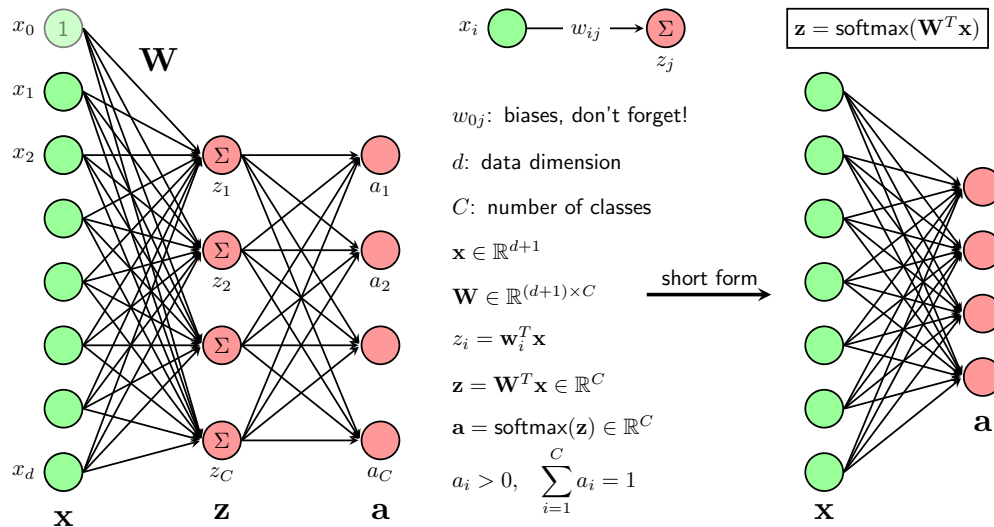
Chú ý rằng z_i có thể nhận giá trị cả âm và dương vì nó là một tổ hợp tuyến tính của các thành phần của vector đặc trưng \mathbf{x} . Một hàm số khả vi đơn giản có thể chắc chắn biến z_i thành một giá trị dương, và hơn nữa, đồng biến, là hàm $\exp(z_i) = e^{z_i}$. Điều kiện khả vi để thuận lợi cho việc sử dụng đạo hàm cho việc tối ưu. Điều kiện cuối cùng, tổng các a_i bằng một có thể được đảm bảo nếu

$$a_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)}, \quad \forall i = 1, 2, \dots, C \quad (15.1)$$

Mối quan hệ này, với mỗi a_i phụ thuộc vào tất cả các z_i , thoả mãn tất cả các điều kiện đã xét: dương, tổng bằng một, giữ được *thứ tự* của z_i . Hàm số này được gọi là *softmax function*. Lúc này, ta có thể coi rằng

$$p(y_k = i | \mathbf{x}_k; \mathbf{W}) = a_i \quad (15.2)$$

Trong đó, $p(y = i | \mathbf{x}; \mathbf{W})$ được hiểu là xác suất để một điểm dữ liệu \mathbf{x} rơi vào lớp thứ i nếu biết tham số mô hình là ma trận trọng số \mathbf{W} . Hình 15.2 thể hiện mô hình softmax regression



Hình 15.2: Mô hình softmax regression dưới dạng neural network.

dưới dạng neural network. Sự khác nhau giữa mô hình này và mô hình one-vs-rest nằm ở chỗ nó có các liên kết giữa mọi node của hai sublayer màu đỏ.

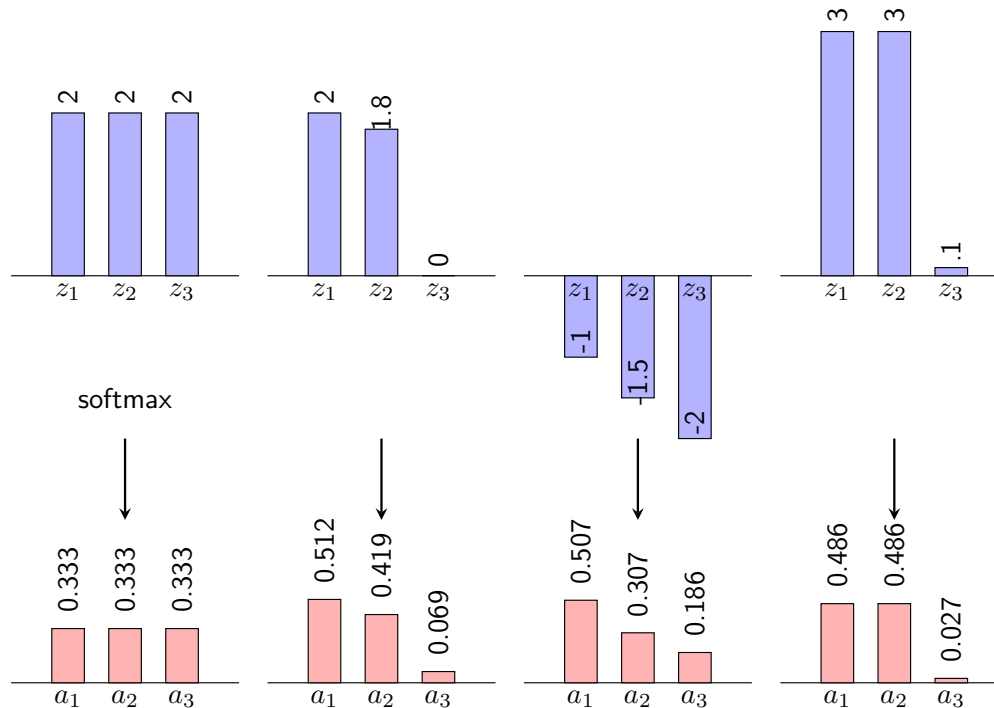
15.2.2 Softmax function trong Python

Dưới đây là một đoạn code thực hiện hàm softmax. Đầu vào là một ma trận với mỗi hàng là một vector \mathbf{z} , đầu ra cũng là một ma trận mà mỗi hàng có giá trị là $\mathbf{a} = \text{softmax}(\mathbf{z})$. Các giá trị của \mathbf{z} còn được gọi là **scores**.

```
import numpy as np
def softmax(Z):
    """
    Compute softmax values for each sets of scores in V.
    each column of V is a set of scores.
    Z: a numpy array of shape (N, C)
    return a numpy array of shape (N, C)
    """
    e_Z = np.exp(Z)
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A
```

15.2.3 Một vài ví dụ

Hình 15.3 mô tả một vài ví dụ về mối quan hệ giữa đầu vào và đầu ra của hàm softmax. Hàng trên màu xanh nhạt thể hiện các scores z_i với giả sử rằng số lớp dữ liệu là ba. Hàng dưới màu đỏ nhạt thể hiện các giá trị đầu ra a_i của hàm softmax.



Hình 15.3: Một số ví dụ về đầu vào và đầu ra của hàm softmax.

Có một vài quan sát như sau:

- Cột 1: Nếu các z_i bằng nhau (bằng 2 hoặc một số bất kỳ), thì các a_i cũng bằng nhau và bằng $1/3$.
- Cột 2: Nếu giá trị lớn nhất trong các z_i là z_1 vẫn bằng 2, thì mặc dù xác suất tương ứng a_1 vẫn là lớn nhất, nó đã thay đổi lên hơn 0.5. Sự chênh lệch ở đầu ra là đáng kể, nhưng thứ tự tương ứng không thay đổi.
- Cột 3: Khi các giá trị z_i là âm thì các giá trị a_i vẫn là dương và thứ tự vẫn được đảm bảo.
- Cột 4: Nếu $z_1 = z_2$, thì $a_1 = a_2$.

Bạn đọc có thể thử với các giá trị khác trực tiếp trên trình duyệt tại <https://goo.gl/pKxQYc>, phần Softmax.

15.2.4 Phiên bản ổn định hơn của softmax function

Khi một trong các z_i quá lớn, việc tính toán $\exp(z_i)$ có thể gây ra hiện tượng *tràn số* (*overflow*), ảnh hưởng lớn tới kết quả của hàm softmax. Có một cách khắc phục hiện tượng này bằng cách dựa trên quan sát

$$\frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)} = \frac{\exp(-c) \exp(z_i)}{\exp(-c) \sum_{j=1}^C \exp(z_j)} = \frac{\exp(z_i - c)}{\sum_{j=1}^C \exp(z_j - c)} \quad (15.3)$$

với c là một hằng số bất kỳ. Vậy một phương pháp đơn giản giúp khắc phục hiện tượng overflow là trừ tất cả các z_i đi một giá trị đủ lớn. Trong thực nghiệm, giá trị đủ lớn này thường được chọn là $c = \max_i z_i$. Vậy chúng ta có thể sửa đoạn code cho hàm `softmax` phía trên bằng cách trừ mỗi hàng của ma trận đầu vào Z đi giá trị lớn nhất trong hàng đó. Ta có phiên bản ổn định hơn là `softmax_stable`¹.

```
def softmax_stable(Z):
    """
    Compute softmax values for each sets of scores in Z.
    each row of Z is a set of scores.
    """
    # Z = Z.reshape(Z.shape[0], -1)
    e_Z = np.exp(Z - np.max(Z, axis = 1, keepdims = True))
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A
```

15.3 Hàm mất mát và phương pháp tối ưu

15.3.1 Cross entropy

Đầu ra của softmax network không còn là một giá trị biểu thị lớp cho dữ liệu mà đã trở thành một vector ở dạng one-hot với chỉ một phần tử bằng 1 tại vị trí tương ứng với lớp đó (tính từ 1), các phần tử còn lại bằng 0.

Với mỗi đầu vào \mathbf{x} , đầu ra tương ứng qua softmax network sẽ là vector $\mathbf{a} = \text{softmax}(\mathbf{W}^T \mathbf{x})$; đầu ra này được gọi là *đầu ra dự đoán*. Trong khi đó, *đầu ra thực sự* của nó là một vector ở dạng one-hot.

Hàm mất mát của softmax regression được xây dựng dựa trên bài toán tối thiểu sự khác nhau giữa *đầu ra dự đoán* \mathbf{a} và *đầu ra thực sự* \mathbf{y} (ở dạng one-hot). Khi cả hai là các vector thể hiện xác suất, khoảng cách giữa chúng thường được đo bằng một đại lượng được gọi là *cross entropy*. Một đặc điểm nổi bật của đại lượng này là nếu cố định một vector xác suất, giá trị của nó *đạt giá trị nhỏ nhất khi hai vector xác suất bằng nhau, và rất lớn khi hai vector đó lệch nhau nhiều*.

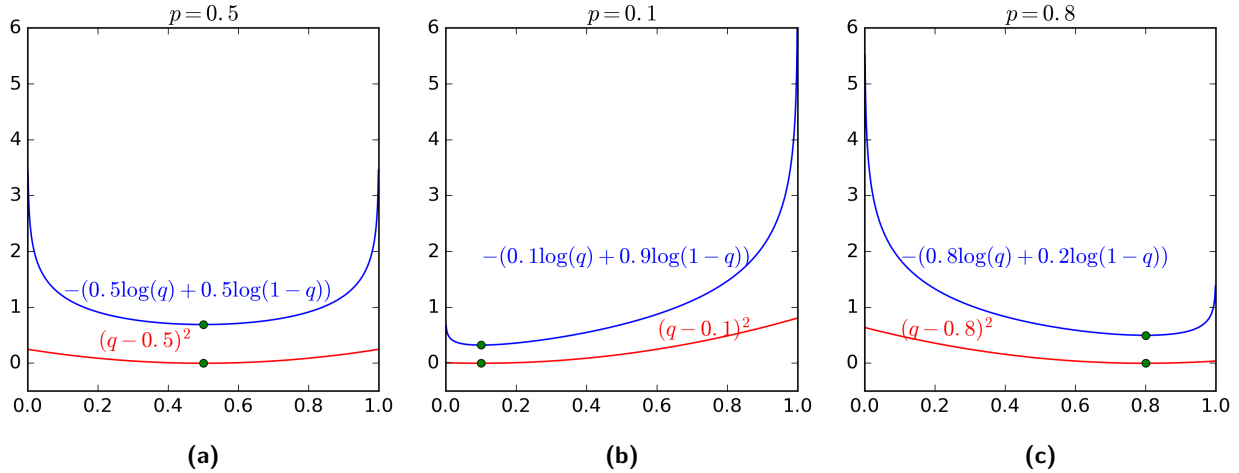
Cross entropy giữa hai vector phân phối \mathbf{p} và \mathbf{q} rời rạc được định nghĩa bởi

$$H(\mathbf{p}, \mathbf{q}) = - \sum_{i=1}^C p_i \log q_i \quad (15.4)$$

Hình 15.4 thể hiện rõ ưu điểm của hàm cross entropy so với hàm bình phương khoảng cách Euclid. Đây là ví dụ trong trường hợp $C = 2$ và p_1 lần lượt nhận các giá trị 0.5, 0.1 và 0.8. Chú ý rằng $p_2 = 1 - p_1$. Có hai nhận xét quan trọng sau đây:

1. Giá trị nhỏ nhất của cả hai hàm số đạt được khi $q = p$ tại hoành độ các điểm màu lục.

¹ Xem thêm về cách xử lý mảng numpy trong Python tại <https://fundaml.com>



Hình 15.4: So sánh giữa hàm cross entropy và hàm bình phương khoảng cách. Các điểm màu lục thể hiện các giá trị nhỏ nhất của mỗi hàm.

- Quan trọng hơn, hàm cross entropy nhận giá trị rất cao (tức mất mát rất cao) khi q ở xa p . Trong khi đó, sự chênh lệch giữa các mất mát ở gần hay xa nghiệm của hàm bình phương khoảng cách $(q - p)^2$ là ít đáng kể hơn. Về mặt tối ưu, hàm cross entropy sẽ cho nghiệm *gần* với p hơn vì những nghiệm ở xa bị *trừng phạt* rất nặng.

Hai tính chất trên đây khiến cho cross entropy được sử dụng rộng rãi khi tính khoảng cách giữa hai phân phối xác suất. Tiếp theo, chúng ta sẽ chứng minh nhận định sau.

Cho $\mathbf{p} \in \mathbb{R}_+^C$ là một vector với các thành phần dương và tổng bằng 1. Bài toán tối ưu

$$\mathbf{q} = \arg \min_{\mathbf{q}} H(\mathbf{p}, \mathbf{q})$$

$$\text{thoả mãn: } \sum_{i=1}^C q_i = 1; q_i > 0$$

có nghiệm $\mathbf{q} = \mathbf{p}$.

Bài toán này có thể giải quyết bằng phương pháp nhân tử Lagrange (xem Phụ lục A).

Lagrangian của bài toán này là

$$\mathcal{L}(q_1, q_2, \dots, q_C, \lambda) = - \sum_{i=1}^C p_i \log(q_i) + \lambda \left(\sum_{i=1}^C q_i - 1 \right)$$

Ta cần giải hệ phương trình

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

Từ phương trình thứ nhất ta có $p_i = \lambda q_i$. Vì vậy, $1 = \sum_{i=1}^C p_i = \lambda \sum_{i=1}^C q_i = \lambda \Rightarrow \lambda = 1$. Điều này tương đương với $q_i = p_i, \forall i$. \square

Chú ý

1. Hàm cross entropy không có tính đối xứng $H(\mathbf{p}, \mathbf{q}) \neq H(\mathbf{q}, \mathbf{p})$. Điều này có thể dễ dàng nhận ra ở việc các thành phần của \mathbf{p} trong công thức (1) có thể nhận giá trị bằng 0, trong khi đó các thành phần của \mathbf{q} phải là dương vì $\log(0)$ không xác định. Chính vì vậy, khi sử dụng cross entropy trong các bài toán phân lớp, \mathbf{p} là đầu ra thực sự vì nó là một vector ở dạng one-hot, \mathbf{q} là đầu ra dự đoán. Trong các thành phần thể hiện xác suất của \mathbf{q} , không có thành phần nào tuyệt đối bằng 1 hoặc tuyệt đối bằng 0 (do hàm exp luôn trả về một giá trị dương).
2. Khi \mathbf{p} là một vector ở dạng one-hot, giả sử chỉ có $p_c = 1$, khi đó biểu thức cross entropy trở thành $-\log(q_c)$. Biểu thức này đạt giá trị nhỏ nhất nếu $q_c = 1$, điều này là không khả thi vì nghiệm này không thuộc miền xác định của bài toán. Tuy nhiên, giá trị cross entropy tiệm cận tới 0 khi q_c tiến đến 1. Điều này xảy ra khi z_c rất rất lớn so với các z_i còn lại.

15.3.2 Xây dựng hàm mất mát

Trong trường hợp có C lớp dữ liệu, *mất mát* giữa đầu ra dự đoán và đầu ra thực sự của một điểm dữ liệu \mathbf{x}_i với label (one-hot) \mathbf{y}_i được tính bởi

$$J_i(\mathbf{W}) \triangleq J(\mathbf{W}; \mathbf{x}_i, \mathbf{y}_i) = - \sum_{j=1}^C y_{ji} \log(a_{ji}) \quad (15.5)$$

với y_{ji} và a_{ji} lần lượt là phần tử thứ j của vector xác suất \mathbf{y}_i và \mathbf{a}_i . Nhắc lại rằng đầu ra \mathbf{a}_i phụ thuộc vào đầu vào \mathbf{x}_i và ma trận trọng số \mathbf{W} . Tới đây, nếu để ý rằng chỉ có đúng một j sao cho $y_{ji} = 1, \forall i$, biểu thức (15.5) chỉ còn lại một số hạng tương ứng với giá trị j đó. Để tránh việc sử dụng quá nhiều ký hiệu, chúng ta giả sử rằng y_i là nhãn của điểm dữ liệu \mathbf{x}_i (các nhãn là các số tự nhiên từ 1 tới C), khi đó j chính bằng y_i . Sau khi có ký hiệu này, ta có thể viết lại

$$J_i(\mathbf{W}) = -\log(a_{y_i, i}) \quad (15.6)$$

với $a_{y_i, i}$ là phần tử thứ y_i của vector \mathbf{a}_i .

Kết hợp tất cả các cặp dữ liệu $\mathbf{x}_i, \mathbf{y}_i, i = 1, 2, \dots, N$, hàm mất mát cho softmax regression được xác định bởi

$$J(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \sum_{i=1}^N \log(a_{y_i, i}) \quad (15.7)$$

Ở đây, ma trận trọng số \mathbf{W} là biến cần tối ưu. Mặc dù hàm mất mát này trông phức tạp, đạo hàm của nó rất gọn. Ta cũng có thể thêm weight decay để tránh overfitting bằng cách cộng thêm một đại lượng tỉ lệ với $\|\mathbf{W}\|_F^2$.

$$\bar{J}(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \left(\sum_{i=1}^N \log(a_{y_i, i}) + \frac{\lambda}{2} \|\mathbf{W}\|_F^2 \right) \quad (15.8)$$

Trong các mục tiếp theo, chúng ta sẽ làm việc với hàm mất mát (15.7). Việc mở rộng cho hàm mất mát với regularization (15.8) không phức tạp vì đạo hàm của số hạng regularized $\frac{\lambda}{2} \|\mathbf{W}\|_F^2$ đơn giản là $\lambda \mathbf{W}$. Hàm mất mát (15.7) có thể được thực hiện trên Python như sau².

```
def softmax_loss(X, y, W):
    """
    W: 2d numpy array of shape (d, C),
        each column corresponding to one output node
    X: 2d numpy array of shape (N, d), each row is one data point
    y: 1d numpy array -- label of each row of X
    """
    A = softmax_stable(X.dot(W))
    id0 = range(X.shape[0]) # indexes in axis 0, indexes in axis 1 are in y
    return -np.mean(np.log(A[id0, y]))
```

Chú ý

1. Khi biểu diễn dưới dạng toán học, mỗi điểm dữ liệu là một cột của ma trận \mathbf{X} ; nhưng khi làm việc với numpy, mỗi điểm dữ liệu được đọc theo `axis = 0` của mảng hai chiều \mathbf{X} . Việc này thống nhất với các thư viện scikit-learn hay tensorflow ở chỗ `X[i]` được dùng để chỉ điểm dữ liệu thứ i , tính từ 0. Tức là, nếu có N điểm dữ liệu trong không gian d chiều thì $\mathbf{X} \in \mathbb{R}^{d \times N}$, nhưng `X.shape == (N, d)`.
2. $\mathbf{W} \in \mathbb{R}^{d \times C}$, `W.shape == (d, c)`.
3. $\mathbf{W}^T \mathbf{X}$ sẽ được biểu diễn bởi `X.dot(W)`, và có `shape == (N, C)`.
4. Khi làm việc với phép nhân ma trận hay mảng nhiều chiều trong numpy, ta luôn nhớ chú ý tới kích thước của các ma trận sao cho các phép nhân thực hiện được.

15.3.3 Tối ưu hàm mất mát

Hàm mất mát sẽ được tối ưu bằng gradient descent, cụ thể là mini-batch gradient descent. Mỗi lần cập nhật của mini-batch gradient descent được thực hiện trên một *batch* có số phần tử $1 < k \ll N$. Để tính được đạo hàm của hàm mất mát theo tập con này, trước hết ta xem xét đạo hàm của hàm mất mát tại một điểm dữ liệu.

Với chỉ một cặp dữ liệu (\mathbf{x}_i, y_i) , ta lại dùng (15.5)

$$\begin{aligned}
 J_i(\mathbf{W}) &= - \sum_{j=1}^C y_{ji} \log(a_{ji}) = - \sum_{j=1}^C y_{ji} \log \left(\frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)} \right) \\
 &= - \sum_{j=1}^C \left(y_{ji} \mathbf{w}_j^T \mathbf{x}_i - y_{ji} \log \left(\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \right) \\
 &= - \sum_{j=1}^C y_{ji} \mathbf{w}_j^T \mathbf{x}_i + \log \left(\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \quad (15.9)
 \end{aligned}$$

² Xem thêm: Truy cập vào nhiều phần tử của mảng hai chiều trong numpy - FundaML <https://goo.gl/SzLDxa>.

Tiếp theo ta sử dụng công thức

$$\nabla_{\mathbf{W}} J_i(\mathbf{W}) = [\nabla_{\mathbf{w}_1} J_i(\mathbf{W}), \nabla_{\mathbf{w}_2} J_i(\mathbf{W}), \dots, \nabla_{\mathbf{w}_C} J_i(\mathbf{W})] \quad (15.10)$$

Trong đó, gradient theo từng cột của \mathbf{w}_j có thể tính được dựa theo (15.9) và quy tắc chuỗi tính gradient

$$\begin{aligned} \nabla_{\mathbf{w}_j} J_i(\mathbf{W}) &= -y_{ji} \mathbf{x}_i + \frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^C \exp(\mathbf{w}_k^T \mathbf{x}_i)} \mathbf{x}_i \\ &= -y_{ji} \mathbf{x}_i + a_{ji} \mathbf{x}_i = \mathbf{x}_i (a_{ji} - y_{ji}) \\ &= e_{ji} \mathbf{x}_i \quad (\text{với } e_{ji} = a_{ji} - y_{ji}) \end{aligned} \quad (15.11)$$

Giá trị $e_{ji} = a_{ji} - y_{ji}$ chính là sự sai khác giữa đầu ra dự đoán và đầu ra thực sự tại thành phần thứ j . Kết hợp (15.10) và (15.11) với $\mathbf{e}_i = \mathbf{a}_i - \mathbf{y}_i$, ta có

$$\nabla_{\mathbf{W}} J_i(\mathbf{W}) = \mathbf{x}_i [e_{1i}, e_{2i}, \dots, e_{Ci}] = \mathbf{x}_i \mathbf{e}_i^T \quad (15.12)$$

$$\Rightarrow \nabla_{\mathbf{W}} J(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i \mathbf{e}_i^T = \frac{1}{N} \mathbf{X} \mathbf{E}^T \quad (15.13)$$

với $\mathbf{E} = \mathbf{A} - \mathbf{Y}$. Công thức tính đạo hàm đơn giản thế này giúp cho cả batch gradient descent, và mini-batch gradient descent đều có thể dễ dàng được áp dụng. Trong trường hợp mini-batch gradient, giả sử kích thước batch là k , ký hiệu $\mathbf{X}_b \in \mathbb{R}^{d \times k}$, $\mathbf{Y}_b \in \{0, 1\}^{C \times k}$, $\mathbf{A}_b \in \mathbb{R}^{C \times k}$ là dữ liệu ứng với một batch, công thức cập nhật cho một batch sẽ là

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\eta}{N_b} \mathbf{X}_b \mathbf{E}_b^T \quad (15.14)$$

với N_b là kích thước của mỗi batch. Hàm số tính đạo hàm theo \mathbf{W} trong Python có thể được thực hiện như sau:

```
def softmax_grad(X, y, W):
    """
    W: 2d numpy array of shape (d, C),
        each column corresponding to one output node
    X: 2d numpy array of shape (N, d), each row is one data point
    y: 1d numpy array -- label of each row of X
    """
    A = softmax_stable(X.dot(W)) # shape of (N, C)
    id0 = range(X.shape[0])
    A[id0, y] -= 1 # A - Y, shape of (N, C)
    return X.T.dot(A) / X.shape[0]
```

Hàm này đã được kiểm chứng lại bằng hàm `check_grad`.

Từ đó, ta có thể viết hàm số huấn luyện softmax regression như sau:

```
def softmax_fit(X, y, W, lr = 0.01, nepoches = 100, tol = 1e-5, batch_size = 10):
    W_old = W.copy()
    ep = 0
    loss_hist = [loss(X, y, W)] # store history of loss
    N = X.shape[0]
    nbatches = int(np.ceil(float(N)/batch_size))
    while ep < nepoches:
        ep += 1
        mix_ids = np.random.permutation(N) # mix data
        for i in range(nbatches):
            # get the i-th batch
            batch_ids = mix_ids[batch_size*i:min(batch_size*(i+1), N)]
            X_batch, y_batch = X[batch_ids], y[batch_ids]
            W -= lr*softmax_grad(X_batch, y_batch, W) # update gradient descent
        loss_hist.append(softmax_loss(X, y, W))
        if np.linalg.norm(W - W_old)/W.size < tol:
            break
        W_old = W.copy()
    return W, loss_hist
```

Cuối cùng là hàm dự đoán nhãn của các điểm dữ liệu mới. Nhãn của một điểm dữ liệu mới được xác định bằng chỉ số của lớp dữ liệu có xác suất rơi vào cao nhất, và cũng chính là chỉ số của score cao nhất.

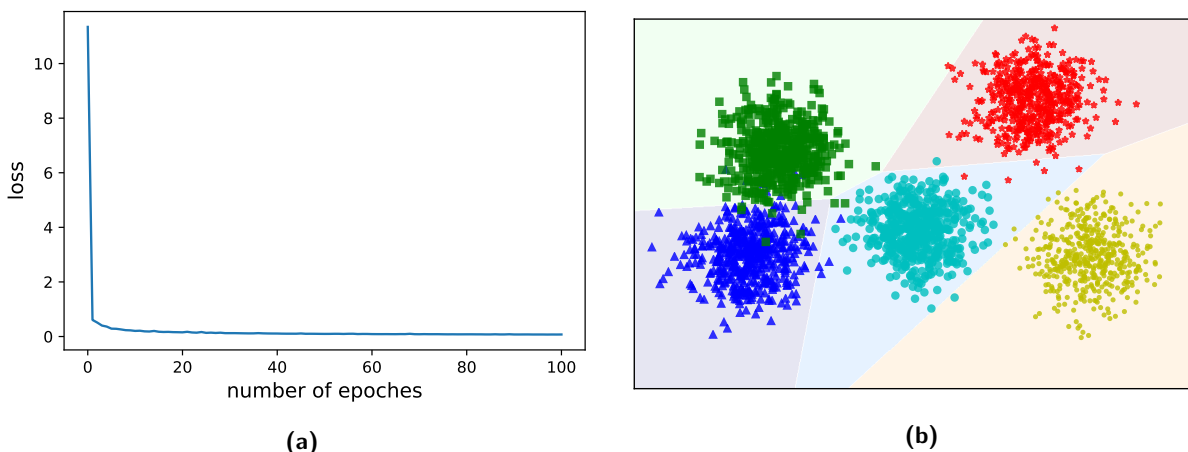
```
def pred(W, X):
    """
    predict output of each columns of X . Class of each x_i is determined by
    location of max probability. Note that classes are indexed from 0.
    """
    return np.argmax(X.dot(W), axis =1)
```

15.4 Ví dụ trên Python

Để minh họa ranh giới của các lớp dữ liệu khi sử dụng softmax regression, chúng ta cùng làm một ví dụ nhỏ trong không gian hai chiều với 5 lớp dữ liệu:

```
C, N = 5, 500 # number of classes and number of points per class
means = [[2, 2], [8, 3], [3, 6], [14, 2], [12, 8]]
cov = [[1, 0], [0, 1]]
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X2 = np.random.multivariate_normal(means[2], cov, N)
X3 = np.random.multivariate_normal(means[3], cov, N)
X4 = np.random.multivariate_normal(means[4], cov, N)
X = np.concatenate((X0, X1, X2, X3, X4), axis = 0) # each row is a datapoint
Xbar = np.concatenate((X, np.ones((X.shape[0], 1))), axis = 1) # bias trick

y = np.asarray([0]*N + [1]*N + [2]*N+ [3]*N + [4]*N)
W_init = np.random.randn(Xbar.shape[1], C)
W, loss_hist = softmax_fit(Xbar, y, W_init, batch_size = 10, nepoches = 100, lr =
    0.05)
```



Hình 15.5: Ví dụ về sử dụng softmax regression cho năm lớp dữ liệu. (a) Nghiệm qua các epochs. (b) Kết quả phân lớp cuối cùng.

Giá trị của hàm mất mát qua các vòng lặp được cho trên Hình 15.5a. Ta thấy rằng hàm mất mát giảm rất nhanh sau đó hội tụ. Các điểm dữ liệu huấn luyện của mỗi lớp là các điểm có màu khác nhau trong Hình 15.5b. Các phần có màu nền khác nhau là các *lãnh thổ* của mỗi lớp dữ liệu tìm được bằng softmax regression. Ta có thể thấy rằng các đường ranh giới có dạng đường thẳng. Kết quả phân chia lãnh thổ cũng khá tốt, chỉ có một số ít điểm trong tập huấn luyện bị phân lớp sai. Để ý thấy rằng dùng softmax regression tốt hơn rất nhiều so với phương pháp kết hợp các bộ phân lớp nhị phân.

MNIST với softmax regression trong scikit-learn

Trong scikit-learn, softmax regression được tích hợp trong class `sklearn.linear_model.LogisticRegression`. Như sẽ thấy trong phần thảo luận, logistic regression chính là softmax regression cho bài toán binary classification. Với bài toán multi-class classification, thư viện này mặc định sử dụng kỹ thuật one-vs-rest. Để sử dụng softmax regression, ta thay đổi thuộc tính `multi_class = 'multinomial'` và `solver = 'lbfgs'`. Ở đây, 'lbfgs' là một phương pháp tối ưu rất mạnh cũng dựa trên đạo hàm. Trong khuôn khổ của cuốn sách, chúng ta sẽ không thảo luận về phương pháp này.

Quay lại với bài toán phân lớp chữ số viết tay trong cơ sở dữ liệu MNIST. Đoạn code dưới đây thực hiện việc lấy ra 10000 điểm dữ liệu trong số 70000 điểm làm tập kiểm thử, còn lại là tập huấn luyện. Bộ phân lớp được sử dụng là softmax regression.

```

import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
mnist = fetch_mldata('MNIST original', data_home='../data/')

X = mnist.data
y = mnist.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=10000)

model = LogisticRegression(C = 1e5,
                           solver = 'lbfgs', multi_class = 'multinomial') # C is inverse of lam
model.fit(X_train, y_train)

y_pred = model.predict(X_test)
print("Accuracy %.2f %" % (100*accuracy_score(y_test, y_pred.tolist())))

```

Kết quả:

Accuracy: 92.19 %

So với kết quả hơn 91.7% của one-vs-rest logistic regression, kết quả của softmax regression đã được cải thiện được một chút. Kết quả thấp như thế này là có thể dự đoán được vì thực ra softmax regression vẫn chỉ tạo ra các đường ranh giới là các đường tuyến tính. Kết quả tốt nhất của bài toán phân loại chữ số trong MNIST hiện nay vào khoảng hơn 99.7%, đạt được bằng một convolutional neural network với rất nhiều hidden layer và layer cuối cùng chính là một softmax regression.

15.5 Thảo luận

15.5.1 Logistic regression là một trường hợp đặc biệt của softmax regression

Khi $C = 2$, softmax regression và logistic regression là giống nhau. Thật vậy, với $C = 2$, đầu ra của hàm softmax cho một đầu vào \mathbf{x} là

$$a_1 = \frac{\exp(\mathbf{w}_1^T \mathbf{x})}{\exp(\mathbf{w}_1^T \mathbf{x}) + \exp(\mathbf{w}_2^T \mathbf{x})} = \frac{1}{1 + \exp((\mathbf{w}_2 - \mathbf{w}_1)^T \mathbf{x})}; \quad a_2 = 1 - a_1 \quad (15.15)$$

Từ đây ta thấy rằng, a_1 có dạng là một hàm sigmoid với vector hệ số $\mathbf{w} = -(\mathbf{w}_2 - \mathbf{w}_1)$. Khi $C = 2$, bạn đọc cũng có thể thấy rằng hàm mất mát của logistic regression và softmax regression là như nhau. Hơn nữa, mặc dù có hai outputs, softmax regression có thể biểu diễn bởi một output vì tổng của hai outputs luôn luôn bằng 1.

Softmax regression còn có các tên gọi khác là multinomial logistic regression, hay maximum entropy classifier. Giống như logistic regression, softmax regression được sử dụng trong các bài toán classification. Các tên gọi này được giữ lại vì vấn đề lịch sử.

15.5.2 Ranh giới tạo bởi softmax regression là một mặt tuyến tính

Thật vậy, dựa vào hàm softmax thì một điểm dữ liệu \mathbf{x} được dự đoán là rơi vào class j nếu $a_j \geq a_k, \forall k \neq j$. Bạn đọc có thể chứng minh được rằng

$$a_j \geq a_k \Leftrightarrow z_j \geq z_k \Leftrightarrow \mathbf{w}_j^T \mathbf{x} \geq \mathbf{w}_k^T \mathbf{x} \Leftrightarrow (\mathbf{w}_j - \mathbf{w}_k)^T \mathbf{x} \geq 0 \quad (15.16)$$

Như vậy, một điểm thuộc lớp thứ j nếu và chỉ nếu $(\mathbf{w}_j - \mathbf{w}_k)^T \mathbf{x} \geq 0, \forall k \neq j$. Như vậy, *lãnh thổ* của mỗi lớp dữ liệu là giao của các nửa không gian. Nói cách khác, đường ranh giới giữa các lớp là các mặt tuyến tính.

15.5.3 Softmax Regression là một trong hai classifiers phổ biến nhất

Softmax regression cùng với multi-class support vector machine (Chương 29) là hai bộ phân lớp phổ biến nhất được dùng hiện nay. Softmax regression đặc biệt được sử dụng nhiều trong các deep neural network với rất nhiều hidden layer. Những layer phía trước có thể được coi như một bộ tạo vector đặc trưng, layer cuối cùng thường là một softmax regression.

15.5.4 Source code

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/XU8ZXm>.

Multilayer neural network và backpropagation

16.1 Giới thiệu

16.1.1 Perceptron cho các hàm logic cơ bản

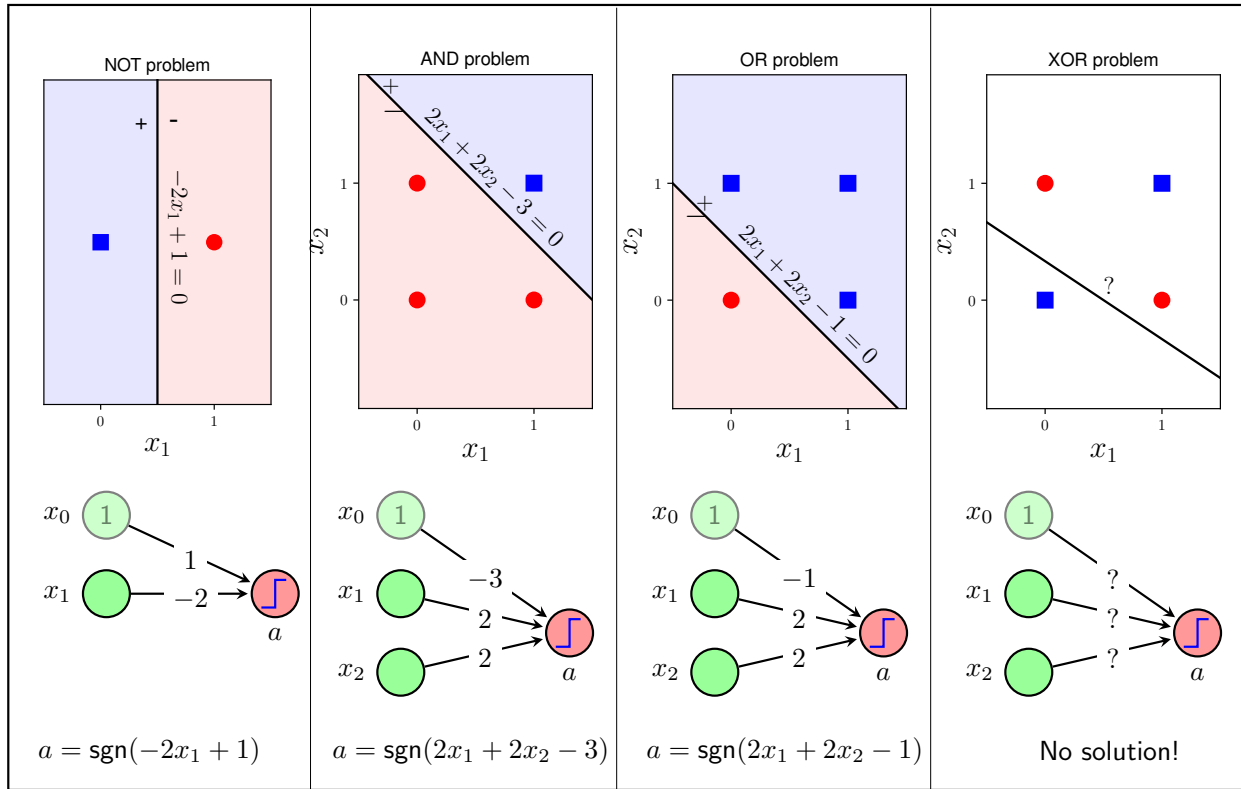
Chúng ta cùng xét khả năng biểu diễn của perceptron (PLA) cho các bài toán biểu diễn các hàm logic nhị phân: NOT, AND, OR, và XOR¹. Để có thể sử dụng perceptron (với đầu ra là 1 hoặc -1), chúng ta sẽ thay các giá trị bằng 0 (false) của tại đầu ra của các hàm này bởi -1. Quan sát hàng trên của Hình 16.1, các điểm hình vuông màu xanh là các điểm có nhãn bằng 1, các điểm hình tròn màu đỏ là các điểm có nhãn bằng -1. Hàng dưới của Hình 16.1 là các mô hình perceptron với các hệ số tương ứng.

Nhận thấy rằng với các bài toán NOT, AND, và OR, dữ liệu hai lớp là linearly separable, vì vậy ta có thể tìm được các hệ số cho perceptron giúp biểu diễn chính xác mỗi hàm số. Chẳng hạn với hàm NOT, khi $x_1 = 0$, ta có $a = \text{sgn}(-2 \times 0 + 1) = 1$; khi $x_1 = 1$, $a = \text{sgn}(-2 \times 1 + 1) = -1$. Trong cả hai trường hợp, đầu ra dự đoán đều giống với đầu ra thực sự. Bạn đọc có thể tự kiểm chứng các hệ số với hàm AND và OR.

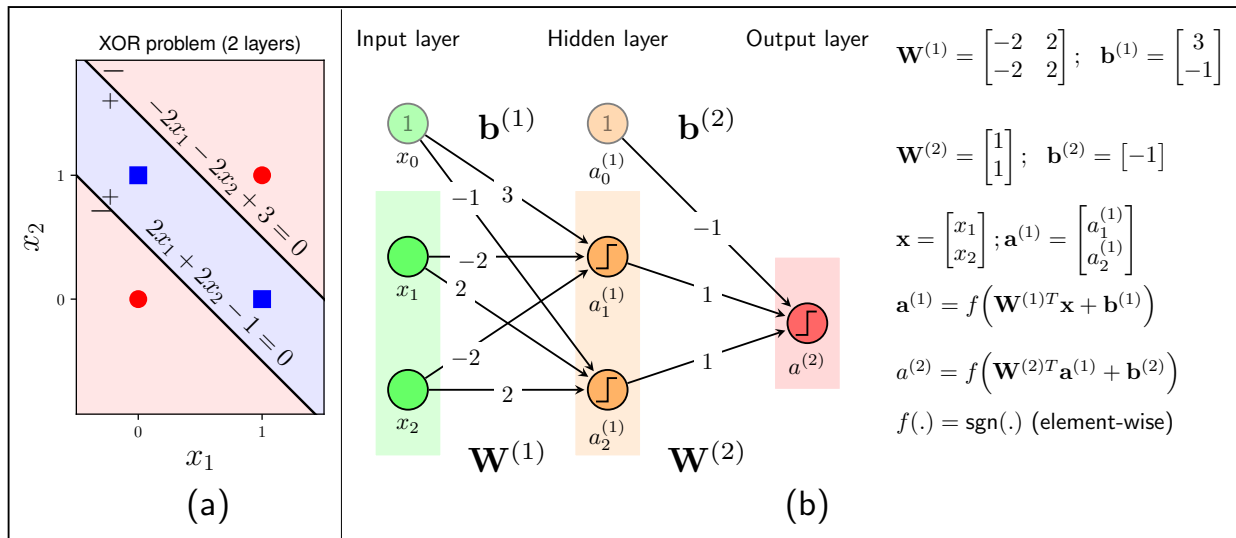
16.1.2 Biểu diễn hàm XOR với nhiều perceptron

Hàm XOR, vì dữ liệu không linearly separable, không thể biểu diễn bằng một perceptron. Nếu thay perceptron bằng logistic regression tức thay hàm kích hoạt từ hàm sign sang hàm *sigmoid*, ta cũng không tìm được các hệ số thỏa mãn, vì về bản chất, logistic regression (hay cả softmax regression) cũng chỉ tạo ra các ranh giới có dạng tuyến tính. Như vậy là các mô hình neural network chúng ta đã biết không thể biểu diễn được hàm số logic đơn giản này.

¹ đầu ra bằng 1 (true) nếu và chỉ nếu hai đầu vào logic khác nhau.



Hình 16.1: Biểu diễn các hàm logic cơ bản sử dụng perceptron learning algorithm.



Hình 16.2: Ba perceptron biểu diễn hàm XOR.

Nhận thấy rằng nếu cho phép sử dụng hai đường thẳng, bài toán biểu diễn hàm XOR có thể được giải quyết như Hình 16.2. Các hệ số tương ứng với hai đường thẳng trong Hình 16.2a được minh họa trên Hình 16.2b bằng các mũi tên xuất phát từ các điểm màu lục và cam. Đầu ra $a_1^{(1)}$ bằng 1 với các điểm nằm về phía (+) của đường thẳng $3 - 2x_1 - 2x_2 = 0$, bằng

-1 với các điểm nằm về phía (-). Tương tự, đầu ra a_2 bằng 1 với các điểm nằm về phía (+) của đường thẳng $-1 + 2x_1 + 2x_2 = 0$. Như vậy, hai đường thẳng ứng với hai perceptron này tạo ra hai đầu ra tại các node $a_1^{(1)}, a_2^{(1)}$. Vì hàm XOR chỉ có một đầu ra nên ta cần làm thêm một bước nữa: coi a_1, a_2 như là đầu vào của một perceptron khác. Trong perceptron mới này, input là các node màu cam (đừng quên bias node luôn có giá trị bằng 1), đầu ra là node màu đỏ. Các hệ số được cho trên Hình 16.2b. Kiểm tra lại một chút, với các điểm hình vuông xanh (Hình 16.2a), $a_1^{(1)} = a_2^{(1)} = 1$, khi đó $a^{(2)} = \text{sgn}(-1 + 1 + 1) = 1$. Với các điểm hình tròn đỏ, vì $a_1^{(1)} = -a_2^{(1)}$ nên $a^{(2)} = \text{sgn}(-1 + a_1^{(1)} + a_2^{(1)}) = \text{sgn}(-1) = -1$. Trong cả hai trường hợp, đầu ra dự đoán đều giống với đầu ra thực sự. Vậy, nếu sử dụng ba perceptron tương ứng với các đầu ra $a_1^{(1)}, a_2^{(1)}, a^{(2)}$, ta sẽ biểu diễn được hàm XOR. Ba perceptron kể trên được xếp vào hai *layers*. Layer thứ nhất: đầu vào - lục, đầu ra - cam. Layer thứ hai: đầu vào - cam, đầu ra - đỏ. Ở đây, đầu ra của layer thứ nhất chính là đầu vào của layer thứ hai. Tổng hợp lại ta được một mô hình mà ngoài layer đầu vào (lục) và đầu ra (đỏ), ta còn có một layer nữa (cam).

Một neural network với nhiều hơn hai layer còn được gọi là *multilayer neural network*, *multilayer perceptrons* (MLPs), *deep feedforward network* hoặc *feedforward neural network*. Từ *feedforward* được hiểu là dữ liệu đi *thẳng* từ đầu vào tới đầu ra theo các mũi tên mà *không quay* lại ở điểm nào, tức là network có dạng một *acyclic graph* (đồ thị không chứa chu trình kín). Tên gọi *perceptron* ở đây có thể gây nhầm lẫn một chút², vì cụm từ này để chỉ neural network với nhiều layer và mỗi layer không nhất thiết, nếu không muốn nói là rất hiếm khi, là một hoặc nhiều perceptron. Hàm kích hoạt có thể là các hàm phi tuyến khác thay vì hàm sgn.

Cụ thể hơn, một multilayer neural network là một neural network có nhiều layer, làm nhiệm vụ xấp xỉ mối quan hệ giữa các cặp quan hệ (\mathbf{x}, \mathbf{y}) trong tập huấn luyện bằng một hàm số có dạng

$$\mathbf{y} \approx g^{(L)} \left(g^{(L-1)} \left(\dots \left(g^{(2)} \left(g^{(1)}(\mathbf{x}) \right) \right) \right) \right), \quad (16.1)$$

Trong đó, layer thứ nhất đóng vai trò như hàm $\mathbf{a}^{(1)} \triangleq g^{(1)}(\mathbf{x})$; layer thứ hai đóng vai trò như hàm $\mathbf{a}^{(2)} \triangleq g^{(2)}(g^{(1)}(\mathbf{x})) = f^{(2)}(\mathbf{a}^{(1)})$, v.v..

Trong phạm vi cuốn sách, chúng ta quan tâm tới các layer đóng vai trò như các hàm có dạng

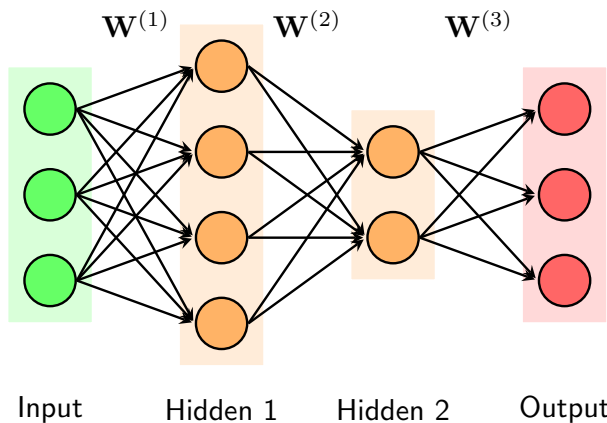
$$g^{(l)}(\mathbf{a}^{(l-1)}) = f^{(l)}(\mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \quad (16.2)$$

với $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$ là ma trận và vector với số chiều phù hợp, $f^{(l)}$ là một hàm số được gọi là *hàm kích hoạt* (*activation function*).

Một vài lưu ý:

- Để cho đơn giản, chúng ta sử dụng ký hiệu $\mathbf{W}^{(l)T}$ để thay cho $(\mathbf{W}^{(l)})^T$ (ma trận chuyển vị). Trong Hình 16.2b, ký hiệu ma trận $\mathbf{W}^{(2)}$ được sử dụng, mặc dù đúng ra nó phải là

² Geoffrey Hinton, *phù thủy Deep Learning*, từng thừa nhận trong khoá học “Neural Networks for Machine Learning” (<https://goo.gl/UfdT1t>) rằng “Multilayer Neural Networks should never have been called Multilayer Perceptron. It is partly my fault, and I’m sorry.”.



Hình 16.3: MLP với hai hidden layers (các biases đã bị ẩn).

vector, để biểu diễn tổng quát cho trường hợp output layer có thể có nhiều hơn một node. Tương tự với bias $\mathbf{b}^{(2)}$.

- Khác với các chương trước về neural network, khi làm việc với multilayer neural network, ta nên tách riêng phần bias và ma trận hệ số. Điều này đồng nghĩa với việc vector input \mathbf{x} là vector KHÔNG mở rộng.

Đầu ra của multilayer neural network loại này ứng với một đầu vào \mathbf{x} có thể được tính theo

$$\mathbf{a}^{(0)} = \mathbf{x} \tag{16.3}$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, 2, \dots, L \tag{16.4}$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), \quad l = 1, 2, \dots, L \tag{16.5}$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \tag{16.6}$$

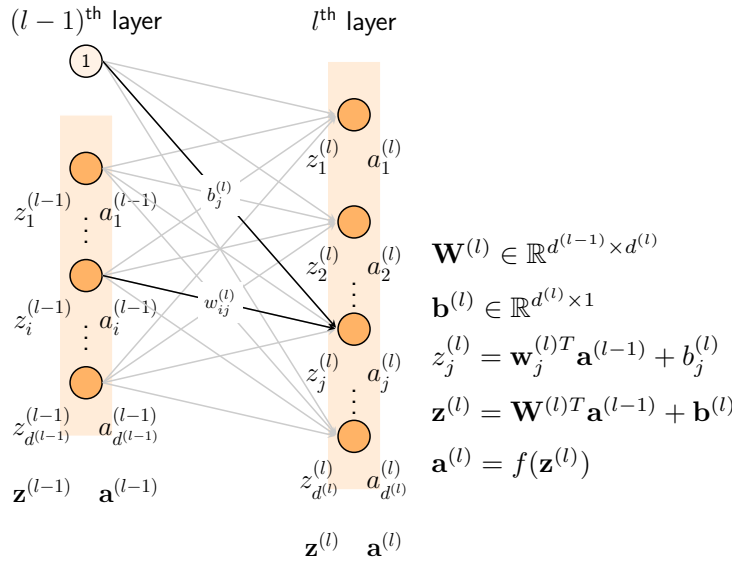
Đây chính là đầu ra dự đoán. Bước này được gọi là *feedforward* vì cách tính toán được thực hiện từ đầu đến cuối của network. Hàm mất mát thỏa mãn đạt giá trị nhỏ khi đầu ra này gần với đầu ra thực sự. Tùy vào bài toán, là classification hoặc regression, chúng ta cần thiết kế các hàm mất mát phù hợp.

16.2 Các ký hiệu và khái niệm

16.2.1 Layer

Ngoài *input layer* và *output layer*, một multilayer neural network có thể có nhiều *hidden layer* ở giữa. Các *hidden layer* theo thứ tự từ input layer đến output layer được đánh số thứ tự là *hidden layer 1*, *hidden layer 2*, v.v.. Hình 16.3 là một ví dụ về một multilayer neural network với hai hidden layer.

Số lượng layer trong một multilayer neural network, được ký hiệu là L , được tính bằng số hidden layer cộng với một. Khi đếm số layer của một multilayer neural network, ta không tính input layer. Trong Hình 16.3, $L = 3$.



Hình 16.4: Các ký hiệu sử dụng trong multilayer neural network.

16.2.2 Units

Quan sát Hình 16.4, mỗi *node* hình tròn trong một layer được gọi là một *unit*. Unit ở input layer, các hidden layer, và output layer được lần lượt gọi là input unit, hidden unit, và output unit. Đầu vào của hidden layer thứ l được ký hiệu bởi $\mathbf{z}^{(l)}$, đầu ra của mỗi unit thường được ký hiệu là $\mathbf{a}^{(l)}$ (thể hiện *activation*, tức giá trị của mỗi unit sau khi ta áp dụng activation function lên đầu vào $\mathbf{z}^{(l)}$). Đầu ra của unit thứ i trong layer thứ l được ký hiệu là $a_i^{(l)}$. Giả sử thêm rằng số unit trong layer thứ l (không tính bias) là $d^{(l)}$. Vector biểu diễn output của layer thứ l được ký hiệu là $\mathbf{a}^{(l)} \in \mathbb{R}^{d^{(l)}}$.

16.2.3 Weights và Biases

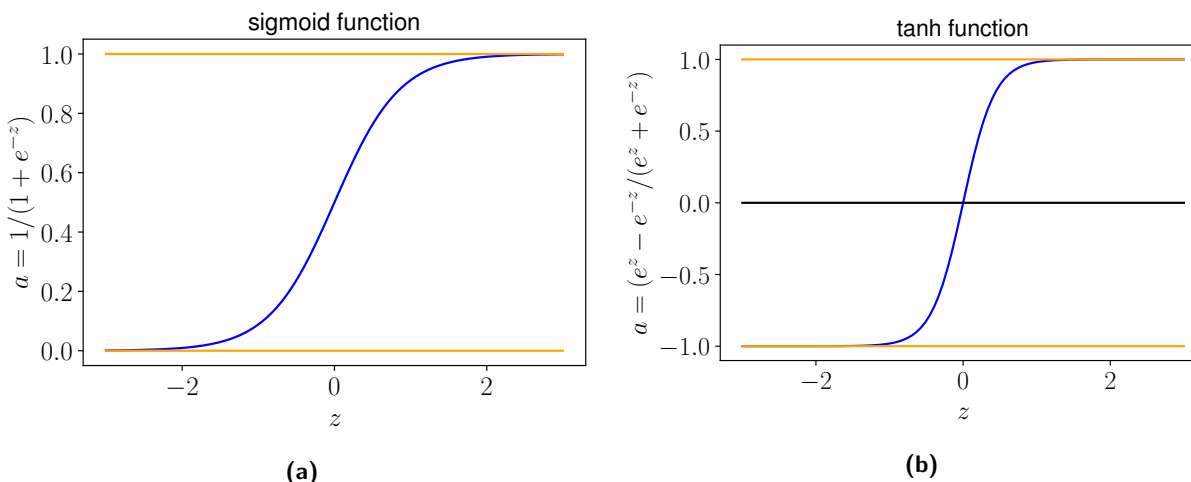
Có L ma trận trọng số cho một multilayer neural network có L layer. Các ma trận này được ký hiệu là $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$, $l = 1, 2, \dots, L$ trong đó $\mathbf{W}^{(l)}$ thể hiện các *kết nối* từ layer thứ $l - 1$ tới layer thứ l (nếu ta coi input layer là layer thứ 0). Cụ thể hơn, phần tử $w_{ij}^{(l)}$ thể hiện kết nối từ node thứ i của layer thứ $(l - 1)$ tới node thứ j của layer thứ l . Các bias của layer thứ l được ký hiệu là $\mathbf{b}^{(l)} \in \mathbb{R}^{d^{(l)}}$. Các trọng số này được ký hiệu như trên Hình 16.4. Khi tối ưu một multilayer neural network cho một công việc nào đó, chúng ta cần đi tìm các weight và bias này. Tập hợp các weight và bias lần lượt được ký hiệu là \mathbf{W} và \mathbf{b} .

16.3 Activation function—Hàm kích hoạt

Mỗi output của một layer (trừ input layer) được tính dựa vào công thức

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}) \tag{16.7}$$

Trong đó $f^{(l)}(\cdot)$ là một hàm kích hoạt phi tuyến. Nếu hàm kích hoạt tại một layer là một hàm tuyến tính, layer này và layer tiếp theo có thể rút gọn thành một layer vì *hợp của các hàm tuyến tính là một hàm tuyến tính*.



Hình 16.5: Ví dụ về đồ thị của hàm (a) sigmoid và (b) tanh.

Hàm kích hoạt thường là một hàm số áp dụng lên *từng phần tử* của ma trận hoặc vector đầu vào, nói cách khác, hàm kích hoạt thường là *element-wise*³.

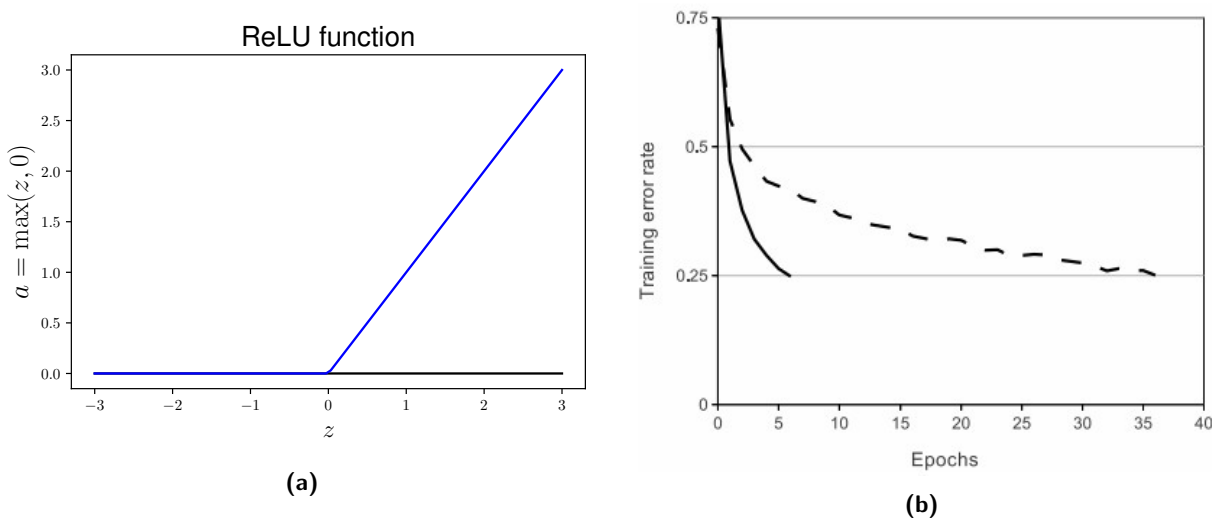
16.3.1 Hàm *sgn* không được sử dụng trong MLP

Hàm *sgn* chỉ được sử dụng trong perceptron. Trong thực tế, hàm *sgn* không được sử dụng vì và đạo hàm tại hầu hết các điểm bằng 0 (trừ tại điểm 0 không có đạo hàm). Việc đạo hàm bằng 0 này khiến cho các thuật toán dựa trên gradient không hoạt động.

16.3.2 Sigmoid và tanh

Hàm *sigmoid* có dạng $\text{sigmoid}(z) = 1/(1 + \exp(-z))$ với đồ thị như trong Hình 16.5a. Nếu đầu vào lớn, hàm số sẽ cho đầu ra gần với 1. Với đầu vào nhỏ (rất âm), hàm số sẽ cho đầu ra gần với 0. Trước đây, hàm kích hoạt này được sử dụng nhiều vì có đạo hàm rất *đẹp*. Những năm gần đây, hàm số này ít khi được sử dụng. Một hàm tương tự thường được sử dụng và mang lại hiệu quả tốt hơn là hàm tanh với $\text{tanh}(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$. Hàm số này có

tính chất đầu ra chạy từ -1 đến 1, khiến cho nó có tính chất zero-centered, thay vì chỉ dương như hàm sigmoid. Gần đây, hàm sigmoid chỉ được sử dụng ở output layer khi yêu cầu của đầu ra là các giá trị nhị phân. Một nhược điểm dễ nhận thấy là khi đầu vào có trị tuyệt đối lớn (rất âm hoặc rất dương), đạo hàm của cả sigmoid và tanh sẽ rất gần với 0. Điều này đồng nghĩa với việc các hệ số tương ứng với unit đang xét sẽ gần như không được cập nhật khi sử dụng công thức cập nhật gradient descent. Thêm nữa, khi khởi tạo các hệ số cho multilayer neural network với hàm kích hoạt sigmoid, chúng ta phải tránh trường hợp đầu vào một hidden layer nào đó quá lớn, vì khi đó đầu ra của hidden layer đó sẽ rất gần với 0 hoặc 1, dẫn đến đạo hàm bằng 0 và gradient descent hoạt động không hiệu quả.



Hình 16.6: Hàm ReLU và tốc độ hội tụ khi so sánh với hàm tanh.

16.3.3 ReLU

ReLU (Rectified Linear Unit) được sử dụng rộng rãi gần đây vì tính đơn giản của nó. Đồ thị của hàm ReLU được minh họa trên Hình 16.6a. Hàm ReLU có công thức toán học $f(z) = \max(0, z)$ - rất đơn giản, rất lợi về mặt tính toán. Đạo hàm của nó bằng 0 tại các điểm âm, bằng 1 tại các điểm dương. ReLU được chứng minh giúp cho việc huấn luyện các multilayer neural network và deep network (rất nhiều hidden layer) nhanh hơn rất nhiều so với hàm tanh [KSH12]. Hình 16.6b so sánh sự hội tụ của hàm mất mát khi sử dụng hai hàm kích hoạt ReLU và tanh. Sự tăng tốc này được cho là vì ReLU được tính toán gần như tức thời và gradient của nó cũng được tính cực nhanh.

Mặc dù cũng có nhược điểm đạo hàm bằng 0 với các giá trị đầu vào âm, ReLU được chứng minh bằng thực nghiệm rằng có thể khắc phục việc này bằng việc tăng số hidden unit⁴. ReLU trở thành hàm kích hoạt đầu tiên chúng ta nên thử khi thiết kế một multilayer neural network. Hầu hết các network đều có hàm kích hoạt là ReLU trong các hidden unit, trừ hàm kích hoạt ở output layer phụ thuộc vào đầu ra thực sự của mỗi bài toán (có thể nhận giá trị âm, hoặc nhị phân, v.v.).

Ngoài ra, các biến thể của ReLU như leaky rectified linear unit (Leaky ReLU), parametric rectified linear unit (PReLU) và randomized leaky rectified linear units (RReLU) [XWCL15] cũng được sử dụng và được báo cáo có kết quả tốt. Trong thực tế, trước khi thiết kế, ta thường không biết chính xác hàm kích hoạt nào sẽ cho kết quả tốt nhất. Tuy nhiên, ta nên bắt đầu bằng ReLU, nếu kết quả chưa khả quan thì có thể thay thế bằng các biến thể của nó và so sánh kết quả.

³ Hàm softmax không phải là một hàm *element-wise* vì nó sử dụng mọi thành phần của vector đầu vào.

⁴ *Neural Networks and Deep Learning – Activation function* (<https://goo.gl/QGjKmU>).

16.4 Backpropagation

Phương pháp phổ biến nhất để tối ưu multilayer neural network chính là gradient descent (GD). Để áp dụng GD, chúng ta cần tính được đạo hàm của hàm mất mát theo từng ma trận trọng số $\mathbf{W}^{(l)}$ và vector bias $\mathbf{b}^{(l)}$.

Giả sử $J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y})$ là một hàm mất mát của bài toán, trong đó \mathbf{W}, \mathbf{b} là tập hợp tất cả các ma trận trọng số giữa các layer và vector bias của mỗi layer. \mathbf{X}, \mathbf{Y} là cặp dữ liệu huấn luyện với mỗi cột tương ứng với một điểm dữ liệu. Để có thể áp dụng các phương pháp gradient descent, chúng ta cần tính được các $\nabla_{\mathbf{W}^{(l)}} J; \nabla_{\mathbf{b}^{(l)}} J, \forall l = 1, 2, \dots, L$.

Nhắc lại quá trình feedforward

$$\mathbf{a}^{(0)} = \mathbf{x} \quad (16.8)$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, 2, \dots, L \quad (16.9)$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), \quad l = 1, 2, \dots, L \quad (16.10)$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \quad (16.11)$$

Một ví dụ của hàm mất mát là hàm mean square error (MSE):

$$J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{n=1}^N \|\mathbf{y}_n - \hat{\mathbf{y}}_n\|_2^2 = \frac{1}{N} \sum_{n=1}^N \|\mathbf{y}_n - \mathbf{a}_n^{(L)}\|_2^2 \quad (16.12)$$

với N là số cặp dữ liệu (\mathbf{x}, \mathbf{y}) trong tập huấn luyện. Theo các công thức này, việc tính toán trực tiếp các giá trị đạo hàm là cực kỳ phức tạp vì hàm mất mát không phụ thuộc trực tiếp vào các ma trận hệ số và vector bias. Phương pháp phổ biến nhất được dùng có tên là backpropagation giúp tính đạo hàm ngược từ layer cuối cùng đến layer đầu tiên. Layer cuối cùng được tính toán trước vì nó gần gũi hơn với đầu ra dự đoán và hàm mất mát. Việc tính toán đạo hàm của các ma trận hệ số trong các layer trước được thực hiện dựa trên một quy tắc chuỗi quen thuộc cho đạo hàm của hàm hợp.

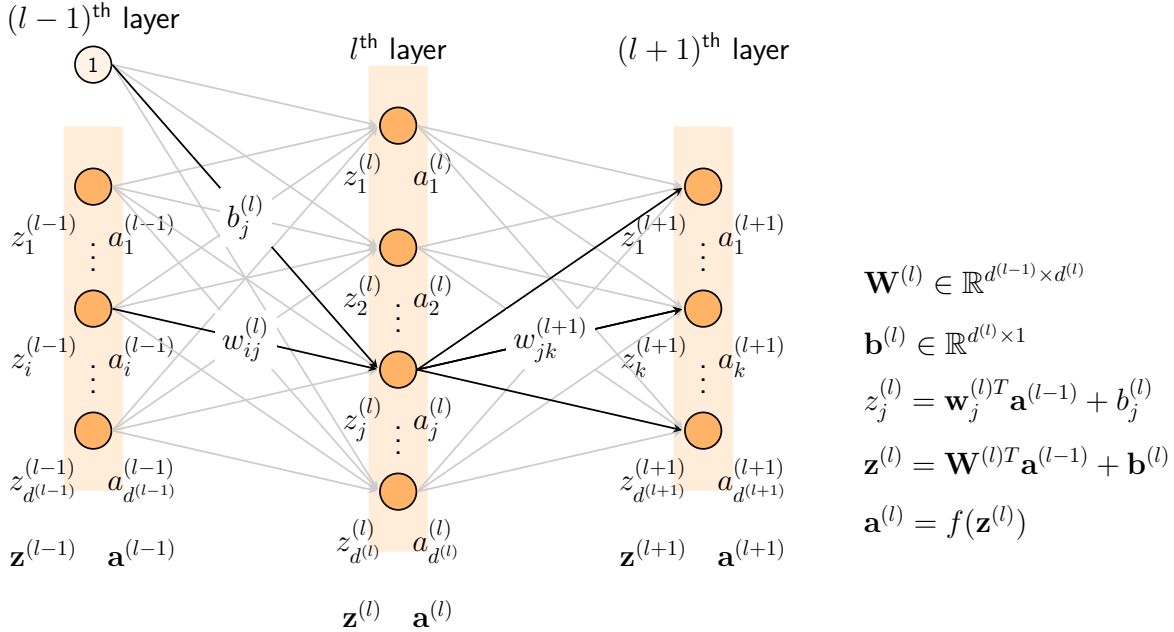
Stochastic gradient descent có thể được sử dụng để tính gradient cho các ma trận trọng số và biases dựa trên một cặp điểm training \mathbf{x}, \mathbf{y} . Để cho đơn giản, ta coi J là hàm mất mát nếu chỉ xét cặp điểm này, ở đây J là hàm mất mát bất kỳ, không chỉ hàm MSE như ở trên. Đạo hàm của hàm mất mát theo chỉ một thành phần của ma trận trọng số của output layer

$$\frac{\partial J}{\partial w_{ij}^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = e_j^{(L)} a_i^{(L-1)} \quad (16.13)$$

trong đó $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$ thường là một đại lượng không quá khó để tính toán và $\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}$

vì $z_j^{(L)} = \mathbf{w}_j^{(L)T} \mathbf{a}^{(L-1)} + b_j^{(L)}$. Tương tự, đạo hàm của hàm mất mát theo bias của layer cuối cùng là

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = e_j^{(L)} \quad (16.14)$$



Hình 16.7: Mô phỏng cách tính backpropagation. Layer cuối có thể là output layer.

Với đạo hàm theo hệ số ở các lớp l thấp hơn, chúng ta hãy xem Hình 16.7. Ở đây, tại mỗi unit, đầu vào z và đầu ra a được viết riêng để chúng ta tiện theo dõi.

Dựa vào Hình 16.7, bằng quy nạp ngược từ cuối, ta có thể tính được

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = e_j^{(l)} a_i^{(l-1)} \tag{16.15}$$

với

$$e_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \tag{16.16}$$

$$= \left(\sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f^{(l)'}(z_j^{(l)}) = \left(\sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f^{(l)'}(z_j^{(l)}) \tag{16.17}$$

trong đó $\mathbf{e}^{(l+1)} = [e_1^{(l+1)}, e_2^{(l+1)}, \dots, e_{d^{(l+1)}}^{(l+1)}]^T \in \mathbb{R}^{d^{(l+1)} \times 1}$ và $\mathbf{w}_{j:}^{(l+1)}$ được hiểu là **hàng** thứ j của ma trận $\mathbf{W}^{(l+1)}$ (chú ý dấu hai chấm, khi không có dấu này, chúng ta mặc định dùng nó để ký hiệu cho vector cột). Dấu \sum tính tổng ở dòng thứ hai trong phép tính trên xuất hiện vì $a_j^{(l)}$ đóng góp vào việc tính tất cả các $z_k^{(l+1)}$, $k = 1, 2, \dots, d^{(l+1)}$. Biểu thức đạo hàm ngoài dấu ngoặc lớn là vì $a_j^{(l)} = f^{(l)}(z_j^{(l)})$. Tới đây, ta có thể thấy rằng việc activation function có đạo hàm đơn giản sẽ có ích rất nhiều trong việc tính toán. Với cách làm tương tự, bạn đọc có thể suy ra

$$\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)}. \tag{16.18}$$

Nhận thấy rằng trong các công thức trên đây, việc tính các $e_j^{(l)}$ đóng một vài trò quan trọng. Hơn nữa, để tính được giá trị này, ta cần tính được các $e_j^{(l+1)}$. Nói cách khác, ta cần tính ngược các giá trị này từ cuối. Cái tên *backpropagation* cũng xuất phát từ việc này.

Việc tính toán các đạo hàm khi sử dụng SGD có thể tóm tắt như sau

Thuật toán 16.1: Backpropagation tới $w_{ij}^{(l)}, b_i^{(l)}$

1. *Bước feedforward*: Với 1 giá trị đầu vào \mathbf{x} , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các giá trị activation $\mathbf{a}^{(l)}$ tại mỗi layer.
2. Với mỗi unit j ở output layer, tính

$$e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}; \quad \frac{\partial J}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} e_j^{(L)}; \quad \frac{\partial J}{\partial b_j^{(L)}} = e_j^{(L)} \quad (16.19)$$

3. Với $l = L - 1, L - 2, \dots, 1$, tính:

$$e_j^{(l)} = \left(\mathbf{w}_{j:}^{(l+1)} \mathbf{e}^{(l+1)} \right) f'(z_j^{(l)}) \quad (16.20)$$

4. Cập nhật đạo hàm cho từng hệ số

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = a_i^{(l-1)} e_j^{(l)}; \quad \frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)} \quad (16.21)$$

Phiên bản *vectorization* của thuật toán trên có thể được thực hiện như sau.

Thuật toán 16.2: Backpropagation tới $\mathbf{W}^{(l)}$ và vector bias $\mathbf{b}^{(l)}$

1. *Bước feedforward*: Với một giá trị đầu vào \mathbf{x} , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các activation $\mathbf{a}^{(l)}$ tại mỗi layer.
2. Với output layer, tính

$$\mathbf{e}^{(L)} = \nabla_{\mathbf{z}^{(L)}} J \in \mathbb{R}^{d^{(L)}}; \quad \nabla_{\mathbf{W}^{(L)}} J = \mathbf{a}^{(L-1)} \mathbf{e}^{(L)T} \in \mathbb{R}^{d^{(L-1)} \times d^{(L)}}; \quad \nabla_{\mathbf{b}^{(L)}} J = \mathbf{e}^{(L)}$$

3. Với $l = L - 1, L - 2, \dots, 1$, tính:

$$\mathbf{e}^{(l)} = \left(\mathbf{W}^{(l+1)} \mathbf{e}^{(l+1)} \right) \odot f'(\mathbf{z}^{(l)}) \in \mathbb{R}^{d^{(l)}} \quad (16.22)$$

trong đó \odot là *element-wise product* hay *Hadamard product* tức lấy từng thành phần của hai vector nhân với nhau để được vector kết quả.

4. Cập nhật đạo hàm cho các ma trận trọng số và vector bias:

$$\nabla_{\mathbf{W}^{(l)}} J = \mathbf{a}^{(l-1)} \mathbf{e}^{(l)T} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}; \quad \nabla_{\mathbf{b}^{(l)}} J = \mathbf{e}^{(l)} \quad (16.23)$$

Khi làm việc với các phép tính đạo hàm phức tạp, ta luôn cần nhớ hai điều sau.

1. Đạo hàm của một hàm có đầu ra là một số vô hướng theo một vector hoặc ma trận là một đại lượng có cùng chiều với vector hoặc ma trận đó.
2. Để các phép nhân các ma trận, vector thực hiện được, ta cần đảm bảo chiều của chúng phù hợp.

Trong công thức $\nabla_{\mathbf{W}^{(L)}} J = \mathbf{a}^{(L-1)} \mathbf{e}^{(L)T}$, vế trái là một ma trận thuộc $\mathbb{R}^{d^{(L-1)} \times d^{(L)}}$, vậy vế phải cũng phải là một đại lượng có chiều tương tự. Từ đó bạn đọc có thể thấy tại sao vế phải phải là $\mathbf{a}^{(L-1)} \mathbf{e}^{(L)T}$ mà không thể là $\mathbf{a}^{(L-1)} \mathbf{e}^{(L)}$ hay $\mathbf{e}^{(L)} \mathbf{a}^{(L-1)}$.

16.4.1 Backpropagation cho batch (mini-batch) gradient descent

Nếu ta muốn thực hiện batch hoặc mini-batch GD thì thế nào? Trong thực tế, mini-batch GD được sử dụng nhiều nhất với các bài toán mà tập huấn luyện lớn. Nếu lượng dữ liệu là nhỏ, batch GD trực tiếp được sử dụng. Khi đó, cặp (input, output) sẽ ở dạng ma trận (\mathbf{X}, \mathbf{Y}) . Giả sử rằng mỗi lần tính toán, ta lấy N dữ liệu để tính toán. Khi đó, $\mathbf{X} \in \mathbb{R}^{d^{(0)} \times N}$, $\mathbf{Y} \in \mathbb{R}^{d^{(L)} \times N}$. Với $d^{(0)} = d$ là chiều của dữ liệu đầu vào (không tính bias).

Khi đó các activation sau mỗi layer sẽ có dạng $\mathbf{A}^{(l)} \in \mathbb{R}^{d^{(l)} \times N}$. Tương tự, $\mathbf{E}^{(l)} \in \mathbb{R}^{d^{(l)} \times N}$. Và ta cũng có thể suy ra công thức cập nhật như sau.

Thuật toán 16.3: Backpropagation tới $\mathbf{W}^{(l)}$ và bias $\mathbf{b}^{(l)}$ (mini-batch)

1. *Bước feedforward:* Với toàn bộ dữ liệu (batch) hoặc một nhóm dữ liệu (mini-batch) đầu vào \mathbf{X} , tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các activation $\mathbf{A}^{(l)}$ tại mỗi layer. Mỗi cột của $\mathbf{A}^{(l)}$ tương ứng với một cột của \mathbf{X} , tức một điểm dữ liệu đầu vào.
2. Với output layer, tính

$$\mathbf{E}^{(L)} = \nabla_{\mathbf{Z}^{(L)}} J; \quad \nabla_{\mathbf{W}^{(L)}} J = \mathbf{A}^{(L-1)} \mathbf{E}^{(L)T}; \quad \nabla_{\mathbf{b}^{(L)}} J = \sum_{n=1}^N \mathbf{e}_n^{(L)} \quad (16.24)$$

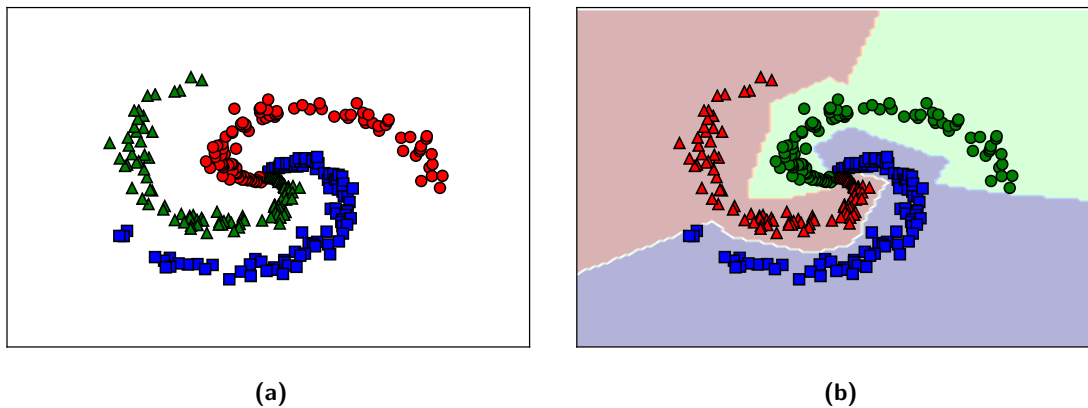
3. Với $l = L - 1, L - 2, \dots, 1$, tính:

$$\mathbf{E}^{(l)} = (\mathbf{W}^{(l+1)} \mathbf{E}^{(l+1)}) \odot f'(\mathbf{Z}^{(l)}) \quad (16.25)$$

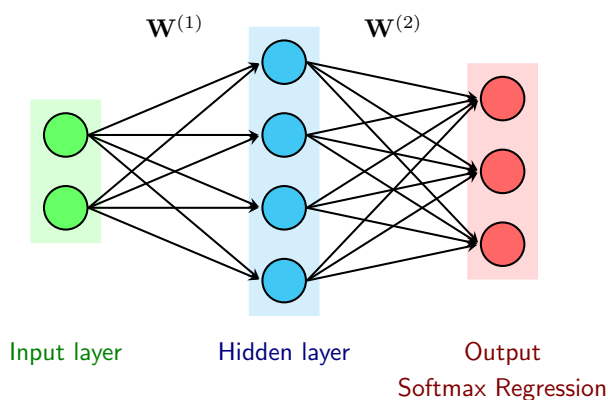
trong đó \odot là element-wise product hay Hadamard product tức lấy từng thành phần của hai ma trận nhân với nhau để được ma trận kết quả.

4. Cập nhật đạo hàm cho ma trận trọng số và vector biases:

$$\nabla_{\mathbf{W}^{(l)}} J = \mathbf{A}^{(l-1)} \mathbf{E}^{(l)T}; \quad \nabla_{\mathbf{b}^{(l)}} J = \sum_{n=1}^N \mathbf{e}_n^{(l)} \quad (16.26)$$



Hình 16.8: Dữ liệu giả trong không gian hai chiều và ví dụ về các ranh giới tốt.



Hình 16.9: Multilayer neural network với input layer có hai unit (bias đã được ẩn), một hidden layer với hàm kích hoạt ReLU (có thể có số lượng hidden unit tùy ý), và output layer là một softmax regression với ba phần tử đại diện cho ba lớp dữ liệu.

16.5 Ví dụ trên Python

Trong mục này, chúng ta sẽ tạo dữ liệu giả trong không gian hai chiều sao cho đường ranh giới giữa các class *không* có dạng tuyến tính. Điều này khiến cho softmax regression không làm việc được. Tuy nhiên, bằng cách thêm một hidden layer, chúng ta sẽ thấy rằng neural network này làm việc rất hiệu quả.

16.5.1 Tạo dữ liệu giả

Các điểm dữ liệu giả của ba lớp được tạo và minh họa bởi các màu khác nhau trên Hình 16.8a. Ta thấy rõ ràng rằng đường ranh giới giữa các lớp dữ liệu không thể là các đường thẳng. Hình 16.8b là một ví dụ về các đường ranh giới được coi là tốt với hầu hết các điểm dữ liệu nằm đúng vào khu vực có màu nền tương ứng. Các đường biên này được tạo sử dụng multilayer neural network với một hidden layer sử dụng ReLU làm hàm kích hoạt và output layer là một softmax regression như trên Hình 16.9. Chúng ta cùng đi sâu vào xây dựng bộ phân lớp dựa trên dữ liệu huấn luyện này.

Nhắc lại hàm ReLU $f(z) = \max(z, 0)$, với đạo hàm

$$f'(z) = \begin{cases} 0 & \text{nếu } z \leq 0 \\ 1 & \text{o.w} \end{cases} \quad (16.27)$$

Vì lượng dữ liệu huấn luyện là nhỏ với 100 điểm cho mỗi lớp, ta có thể dùng batch GD để cập nhật các ma trận hệ số và vector bias. Trước hết, ta cần tính đạo hàm của hàm mất mát theo các ma trận và vector này bằng cách áp dụng backpropagation.

16.5.2 Tính toán Feedforward

Giả sử các cặp dữ liệu huấn luyện là $(\mathbf{x}_i, \mathbf{y}_i)$ với \mathbf{y}_i là một vector ở dạng one-hot. Các điểm dữ liệu này xếp cạnh nhau tạo thành các ma trận đầu vào \mathbf{X} và ma trận đầu ra \mathbf{Y} . Bước feedforward của neural network này được thực hiện như sau.

$$\mathbf{Z}^{(1)} = \mathbf{W}^{(1)T} \mathbf{X} + \mathbf{B}^{(1)} \quad (16.28)$$

$$\mathbf{A}^{(1)} = \max(\mathbf{Z}^{(1)}, \mathbf{0}) \quad (16.29)$$

$$\mathbf{Z}^{(2)} = \mathbf{W}^{(2)T} \mathbf{A}^{(1)} + \mathbf{B}^{(2)} \quad (16.30)$$

$$\hat{\mathbf{Y}} = \mathbf{A}^{(2)} = \text{softmax}(\mathbf{Z}^{(2)}) \quad (16.31)$$

Trong đó $\mathbf{B}^{(1)}, \mathbf{B}^{(2)}$ là các ma trận bias với tất cả các cột bằng nhau và lần lượt bằng $\mathbf{b}^{(1)}$ và $\mathbf{b}^{(2)}$ ⁵. Hàm mất mát được tính dựa trên hàm cross-entropy

$$J \triangleq J(\mathbf{W}, \mathbf{b}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{ji} \log(\hat{y}_{ji}) \quad (16.32)$$

16.5.3 Tính toán Backpropagation

Áp dụng Thuật toán 16.3, ta có

$$\mathbf{E}^{(2)} = \nabla_{\mathbf{Z}^{(2)}} = \frac{1}{N} (\mathbf{A}^{(2)} - \mathbf{Y}) \quad (16.33)$$

$$\nabla_{\mathbf{W}^{(2)}} = \mathbf{A}^{(1)} \mathbf{E}^{(2)T}; \quad \nabla_{\mathbf{b}^{(2)}} = \sum_{n=1}^N \mathbf{e}_n^{(2)} \quad (16.34)$$

$$\mathbf{E}^{(1)} = (\mathbf{W}^{(2)} \mathbf{E}^{(2)}) \odot f'(\mathbf{Z}^{(1)}) \quad (16.35)$$

$$\nabla_{\mathbf{W}^{(1)}} = \mathbf{A}^{(0)} \mathbf{E}^{(1)T} = \mathbf{X} \mathbf{E}^{(1)T}; \quad \nabla_{\mathbf{b}^{(1)}} = \sum_{n=1}^N \mathbf{e}_n^{(1)} \quad (16.36)$$

Các công thức toán học phức tạp này sẽ được lập trình một cách đơn giản hơn trên numpy.

⁵ Ta cần xếp các vector bias giống nhau để tạo thành các ma trận bias vì trong toán học, không có định nghĩa tổng của một ma trận và một vector. Khi lập trình, việc này là khả thi.

16.5.4 Triển khai thuật toán trên numpy

Trước hết, ta viết lại hàm softmax và cross-entropy. Sau đó viết các hàm khởi tạo và dự đoán nhãn của các điểm dữ liệu.

```
def softmax_stable(Z):
    """
    Compute softmax values for each sets of scores in Z.
    each ROW of Z is a set of scores.
    """
    e_Z = np.exp(Z - np.max(Z, axis = 1, keepdims = True))
    A = e_Z / e_Z.sum(axis = 1, keepdims = True)
    return A

def crossentropy_loss(Yhat, y):
    """
    Yhat: a numpy array of shape (Npoints, nClasses) -- predicted output
    y: a numpy array of shape (Npoints) -- ground truth.
    NOTE: We don't need to use the one-hot vector here since most of elements
    are zeros. When programming in numpy, in each row of Yhat, we need to access
    to the corresponding index only.
    """
    id0 = range(Yhat.shape[0])
    return -np.mean(np.log(Yhat[id0, y]))

def mlp_init(d0, d1, d2):
    """
    Initialize W1, b1, W2, b2
    d0: dimension of input data
    d1: number of hidden unit
    d2: number of output unit = number of classes
    """
    W1 = 0.01*np.random.randn(d0, d1)
    b1 = np.zeros(d1)
    W2 = 0.01*np.random.randn(d1, d2)
    b2 = np.zeros(d2)
    return (W1, b1, W2, b2)

def mlp_predict(X, W1, b1, W2, b2):
    """
    Suppose that the network has been trained, predict class of new points.
    X: data matrix, each ROW is one data point.
    W1, b1, W2, b2: learned weight matrices and biases
    """
    Z1 = X.dot(W1) + b1 # shape (N, d1)
    A1 = np.maximum(Z1, 0) # shape (N, d1)
    Z2 = A1.dot(W2) + b2 # shape (N, d2)
    return np.argmax(Z2, axis=1)
```

Tiếp theo là hàm chính huấn luyện softmax regression.

```

def mlp_fit(X, y, W1, b1, W2, b2, eta):
    loss_hist = []
    for i in xrange(20000): # number of epoches
        # feedforward
        Z1 = X.dot(W1) + b1      # shape (N, d1)
        A1 = np.maximum(Z1, 0)   # shape (N, d1)
        Z2 = A1.dot(W2) + b2     # shape (N, d2)
        Yhat = softmax_stable(Z2) # shape (N, d2)
        if i % 1000 == 0: # print loss after each 1000 iterations
            loss = crossentropy_loss(Yhat, y)
            print("iter %d, loss: %f" % (i, loss))
            loss_hist.append(loss)

        # back propagation
        id0 = range(Yhat.shape[0])
        Yhat[id0, y] -= 1
        E2 = Yhat/N              # shape (N, d2)
        dW2 = np.dot(A1.T, E2)   # shape (d1, d2)
        db2 = np.sum(E2, axis = 0) # shape (d2,)
        E1 = np.dot(E2, W2.T)    # shape (N, d1)
        E1[Z1 <= 0] = 0          # gradient of ReLU, shape (N, d1)
        dW1 = np.dot(X.T, E1)    # shape (d0, d1)
        db1 = np.sum(E1, axis = 0) # shape (d1,)

        # Gradient Descent update
        W1 += -eta*dW1
        b1 += -eta*db1
        W2 += -eta*dW2
        b2 += -eta*db2
    return (W1, b1, W2, b2, loss_hist)

```

Sau khi đã hoàn thành các hàm chính của multilayer neural network này, chúng ta đưa dữ liệu vào, xác định số hidden unit, và huấn luyện network.

```

# suppose X, y are training input and output, respectively
d0 = 2      # data dimension
d1 = h = 100 # number of hidden units
d2 = C = 3  # number of classes
eta = 1     # learning rate
(W1, b1, W2, b2) = mlp_init(d0, d1, d2)
(W1, b1, W2, b2, loss_hist) = mlp_fit(X, y, W1, b1, W2, b2, eta)
y_pred = mlp_predict(X, W1, b1, W2, b2)
acc = 100*np.mean(y_pred == y)
print('training accuracy: %.2f %%' % acc)

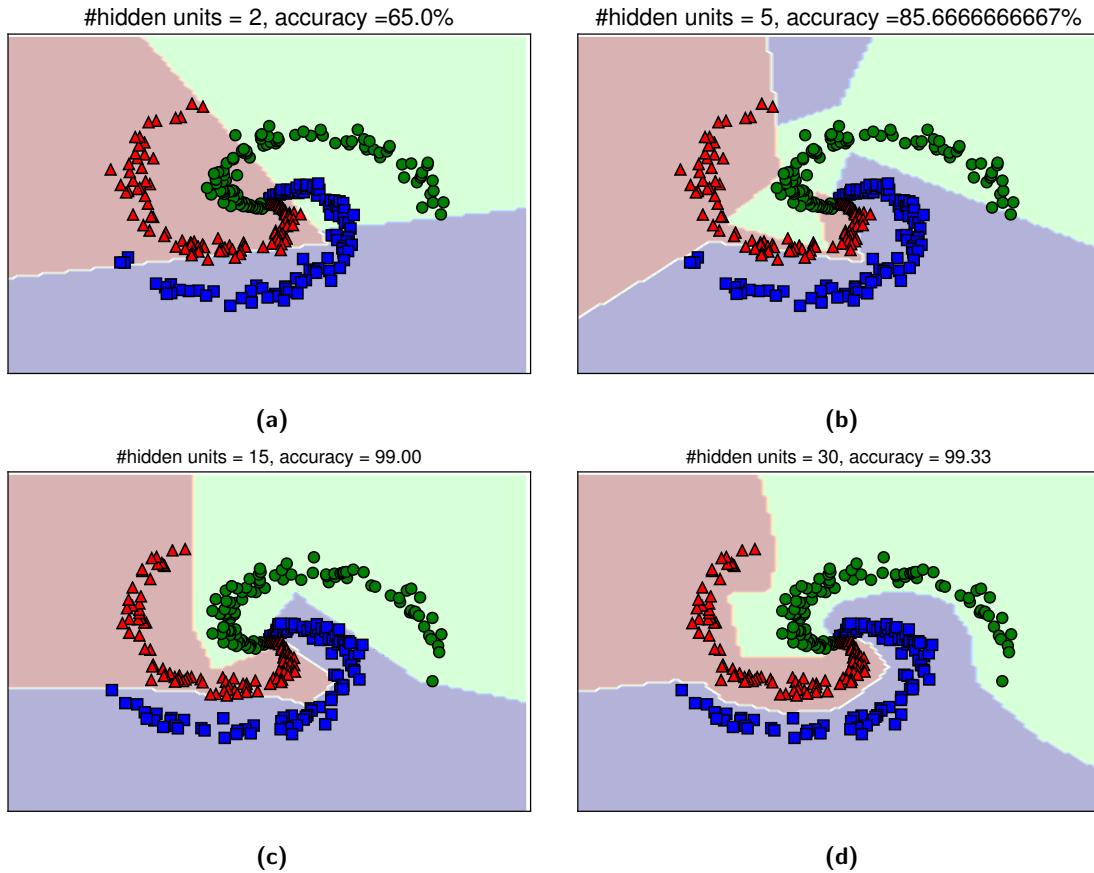
```

Kết quả:

```

iter 0, loss: 1.098628
iter 2000, loss: 0.030014
iter 4000, loss: 0.021071
iter 6000, loss: 0.018158
iter 8000, loss: 0.016914
training accuracy: 99.33 %

```



Hình 16.10: Kết quả với số lượng units trong hidden layer là khác nhau.

Ta có thể thấy rằng hàm mất mát giảm dần và hội tụ. Kết quả phân lớp trên tập huấn luyện rất tốt, chỉ một vài điểm bị phân lớp lỗi, nhiều khả năng chúng nằm ở khu vực trung tâm. Với chỉ một hidden layer, network đã thực hiện công việc gần như hoàn hảo.

Bằng cách thay đổi số lượng hidden unit (biến **d1**) và huấn luyện lại các network, minh họa ranh giới giữa các lớp dữ liệu, chúng ta thu được các kết quả như trên Hình 16.10. Khi chỉ có hai hidden unit, các đường ranh giới vẫn gần như đường thẳng, kết quả là có tới 35% số điểm dữ liệu trong tập huấn luyện bị phân lớp lỗi. Khi số lượng hidden unit là 5, độ chính xác được cải thiện thêm khoảng 20%, tuy nhiên, các đường ranh giới vẫn chưa thực sự tốt. Thậm chí lớp đỏ và lam còn bị chia cắt một cách không tự nhiên. Nếu tiếp tục tăng số lượng hidden unit, ta thấy rằng các đường ranh giới tương đối hoàn hảo.

Có thể chứng minh được rằng với một hàm số liên tục bất kỳ $f(x)$ và một số $\epsilon > 0$, luôn luôn tồn tại một neural network với đầu ra có dạng $g(x)$ với một hidden layer (với số hidden unit đủ lớn và hàm kích hoạt phi tuyến phù hợp) sao cho với mọi $x, |f(x) - g(x)| < \epsilon$. Nói cách khác, neural network có khả năng xấp xỉ bất kỳ hàm liên tục nào [Cyb89].

Trên thực tế, việc tìm ra số lượng hidden unit và hàm kích hoạt nói trên hầu như bất khả thi. Thay vào đó, thực nghiệm chứng minh rằng neural network với nhiều hidden layer kết

hợp với các hàm kích hoạt đơn giản, ví dụ ReLU, có khả năng xấp xỉ dữ liệu tốt hơn tốt hơn. Tuy nhiên, khi số lượng hidden layer lớn lên, số lượng hệ số cần tối ưu cũng lớn lên và mô hình sẽ trở nên phức tạp. Sự phức tạp này ảnh hưởng tới hai khía cạnh. Thứ nhất, tốc độ tính toán sẽ bị chậm đi rất nhiều. Thứ hai, nếu mô hình quá phức tạp, nó có thể biểu diễn rất tốt dữ liệu huấn luyện, nhưng có thể không biểu diễn tốt dữ liệu kiểm thử. Đây chính là hiện tượng overfitting.

Vậy có các kỹ thuật nào giúp tránh overfitting cho multilayer neural network? Ngoài kỹ thuật toán cross-validation, chúng ta quan tâm hơn tới các phương pháp regularization. Các neural network với regularization được gọi là *regularized neural network*. Kỹ thuật phổ biến nhất được dùng để tránh overfitting là *weight decay*.

16.6 Tránh overfitting cho neural network bằng weight decay

Với weight decay, hàm mất mát sẽ được cộng thêm một đại lượng regularization có dạng

$$\lambda R(\mathbf{W}) = \lambda \sum_{l=1}^L \|\mathbf{W}^{(l)}\|_F^2$$

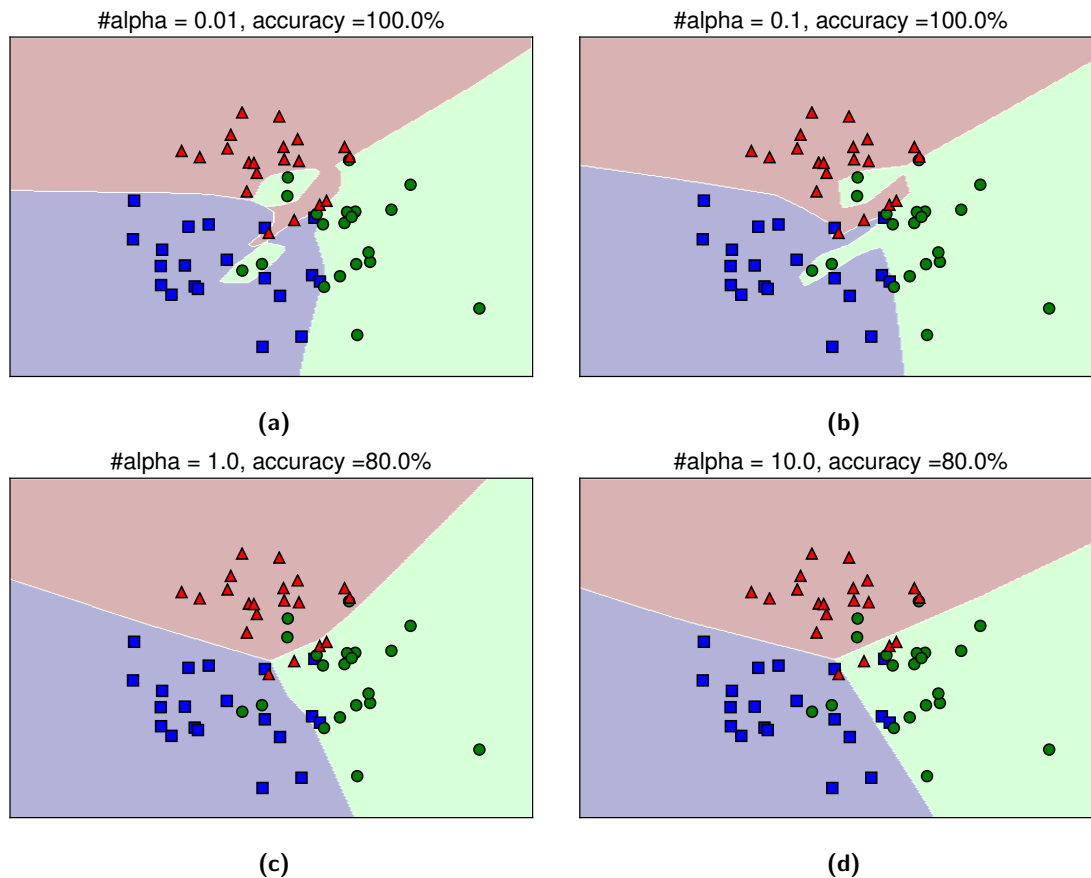
tức tổng bình phương Frobenius norm của tất cả các ma trận hệ số. Chú ý rằng khi làm việc với multilayer neural network, bias hiếm khi được *regularized*. Đây cũng là lý do vì sao ta nên tách rời ma trận hệ số và vector bias khi làm việc với multilayer neural network. Việc tối thiểu hàm mất mát mới (với số hạng regularization) sẽ khiến cho các thành phần của các vector hệ số $\mathbf{W}^{(l)}$ không quá lớn, thậm chí nhiều thành phần sẽ gần với không. Điều này khiến cho việc có nhiều hidden unit vẫn an toàn vì nhiều trong số chúng gần với không.

Tiếp theo, chúng ta sẽ làm một ví dụ nữa trong không gian hai chiều. Lần này, chúng ta sẽ sử dụng thư viện scikit-learn.

```
from __future__ import print_function
import numpy as np
from sklearn.neural_network import MLPClassifier
means = [[-1, -1], [1, -1], [0, 1]]
cov = [[1, 0], [0, 1]]
N = 20
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X2 = np.random.multivariate_normal(means[2], cov, N)

X = np.concatenate((X0, X1, X2), axis = 0)
y = np.asarray([0]*N + [1]*N + [2]*N)

alpha = 1e-1 # regularization parameter
clf = MLPClassifier(solver='lbfgs', alpha=alpha, hidden_layer_sizes=(100))
clf.fit(X, y)
y_pred = clf.predict(X)
acc = 100*np.mean(y_pred == y)
print('training accuracy: %.2f %%' % acc)
```



Hình 16.11: Kết quả với số lượng units trong hidden layer là khác nhau.

Kết quả:

```
training accuracy: 100.00 %
```

Trong đoạn code trên, thuộc tính `alpha` chính là tham số regularization λ . `alpha` càng lớn sẽ khiến các thành phần trong các ma trận hệ số càng nhỏ. Thuộc tính `hidden_layer_sizes` chính là số lượng hidden unit trong mỗi hidden layer. Nếu có nhiều hidden layer, chẳng hạn hai với số lượng hidden unit lần lượt là 10 và 100, ta cần khai báo `hidden_layer_sizes=(10, 100)`. Hình 16.11 minh họa ranh giới giữa các lớp tìm được với các giá trị `alpha` khác nhau, tức mức độ regularization khác nhau. Khi `alpha` nhỏ cỡ 0.01, các ranh giới tìm được trông không được tự nhiên và vùng xác định lớp màu lục không được liên tục. Mặc dù độ chính xác trên tập huấn luyện này là 100%, ta có thể quan sát thấy rằng overfitting đã xảy ra. Với `alpha = 0.1`, kết quả cho thấy *lãnh thổ* của các lớp đã liên tục, nhưng overfitting vẫn xảy ra. Khi `alpha` cao hơn, độ chính xác đã giảm xuống nhưng các đường ranh giới tự nhiên hơn. Bạn đọc có thể thay đổi các giá trị `alpha` trong source code (<https://goo.gl/czxrSf>) và quan sát các hiện tượng xảy ra. Đặc biệt, khi `alpha = 100`, độ chính xác còn 33.33%. Tại sao lại như vậy? Hy vọng bạn đọc có thể tự trả lời được.

16.7 Đọc thêm

1. *Neural Networks: Setting up the Architecture*, Andrej Karpathy (<https://goo.gl/rfzCVK>).
2. *Neural Networks, Case study*, Andrej Karpathy (<https://goo.gl/3ihCxL>).
3. *Lecture Notes on Sparse Autoencoders*, Andrew Ng (<https://goo.gl/yTgtLe>).
4. *Yes you should understand backprop* (<https://goo.gl/8B3h1b>).
5. *Backpropagation, Intuitions*, Andrej Karpathy (<https://goo.gl/fjHzNV>).
6. *How the backpropagation algorithm works*, Michael Nielsen (<https://goo.gl/mwz2kU>).

Recommendation systems—Hệ thống khuyến nghị

Các bạn có lẽ đã gặp những hiện tượng sau đây nhiều lần. Youtube tự động chạy các clip liên quan đến clip bạn đang xem, hoặc tự động gợi ý những clip mà có thể bạn sẽ thích. Khi bạn mua một món hàng trên Amazon, hệ thống sẽ tự động gợi ý những sản phẩm *frequently bought together*, hoặc nó biết bạn có thể thích món hàng nào dựa trên lịch sử mua hàng của bạn. Facebook hiển thị quảng cáo những sản phẩm có liên quan đến từ khoá bạn vừa tìm kiếm trên Google. Facebook gợi ý kết bạn. Netflix tự động gợi ý phim cho người dùng. Và còn rất nhiều ví dụ khác mà hệ thống có khả năng tự động gợi ý cho người dùng những sản phẩm họ *có thể thích*. Bằng cách *quảng cáo hướng đúng đối tượng* đó, hiệu quả của việc marketing cũng sẽ tăng lên.

Những thuật toán đằng sau những ứng dụng này là những thuật toán machine learning có tên gọi chung là *hệ thống khuyến nghị* (*recommender system* hoặc *recommendation system*).

Trong phần này của cuốn sách, chúng ta sẽ cùng tìm hiểu ba thuật toán cơ bản nhất trong rất nhiều các thuật toán *recommendation system*.

Content-based recommendation system

17.1 Giới thiệu

Recommendation system là một mảng khá rộng của machine learning, và có *tuổi đời* ít hơn so với classification hay regression vì internet mới chỉ thực sự bùng nổ khoảng 10-15 năm gần đây. Có hai thực thể chính trong một recommendation system là *user* và *item*. *User* là *người dùng*; *item* là *sản phẩm*, ví dụ như các bộ phim, bài hát, cuốn sách, clip, hoặc cũng có thể là các người dùng khác trong bài toán gợi ý kết bạn. Mục đích chính của các recommender system là dự đoán *mức độ quan tâm* của một người dùng tới một sản phẩm nào đó, qua đó có chiến lược *recommendation* phù hợp.

17.1.1 Hiện tượng *long tail* trong thương mại

Chúng ta cùng đi vào việc so sánh điểm khác nhau căn bản giữa các *cửa hàng thực* và các *cửa hàng điện tử*, xét trên khía cạnh lựa chọn sản phẩm để quảng bá. Ở đây, chúng ta tạm quên đi khía cạnh *có cảm giác thật chạm vào sản phẩm* của các cửa hàng thực. Hãy cùng tập trung vào phần làm thế nào để quảng bá đúng sản phẩm tới đúng khách hàng.

Có thể các bạn đã biết tới *Nguyên lý Pareto (hay quy tắc 20/80)* (<https://goo.gl/NujWjH>): *phần lớn kết quả được gây ra bởi phần nhỏ nguyên nhân*. Phần lớn số từ sử dụng hàng ngày chỉ là một phần nhỏ số từ trong bộ từ điển. Phần lớn của cải được sở hữu bởi phần nhỏ số người. Khi làm thương mại cũng vậy, những sản phẩm bán chạy nhất chỉ chiếm phần nhỏ tổng số sản phẩm.

Các *cửa hàng thực* thường có hai khu vực, một là khu trưng bày, hai là kho. Nguyên tắc dễ thấy để đạt doanh thu cao là trưng ra các sản phẩm phổ biến nhất ở những nơi dễ nhìn thấy và cất những sản phẩm ít phổ biến trong kho. Cách làm này có một hạn chế rõ rệt: những sản phẩm được trưng ra mang tính phổ biến chứ chưa chắc đã phù hợp với một khách hàng

cụ thể. Một cửa hàng có thể có món hàng một khách hàng tìm kiếm nhưng có thể không bán được vì khách hàng không nhìn thấy sản phẩm đó trên giá; việc này dẫn đến việc khách hàng không tiếp cận được sản phẩm ngay cả khi chúng đã được trưng ra. Ngoài ra, vì không gian có hạn, cửa hàng không thể trưng ra tất cả các sản phẩm mà mỗi loại chỉ đưa ra một số lượng nhỏ. Ở đây, phần lớn doanh thu (80%) đến từ phần nhỏ số sản phẩm phổ biến nhất (20%). Nếu sắp xếp các sản phẩm của cửa hàng theo doanh số từ cao đến thấp, ta sẽ nhận thấy có thể phần nhỏ các sản phẩm tạo ra phần lớn doanh số; và một danh sách dài phía sau chỉ tạo ra một lượng đóng góp nhỏ. Hiện tượng này còn được gọi là *long tail phenomenon*, tức phần *đuôi dài* của những sản phẩm ít phổ biến.

Với các *cửa hàng điện tử*, nhược điểm trên hoàn toàn có thể tránh được. Vì *gian trưng bày* của các *cửa hàng điện tử* gần như là vô tận, mọi sản phẩm đều có thể được trưng ra. Hơn nữa, việc sắp xếp online là linh hoạt, tiện lợi với chi phí chuyển đổi gần như bằng 0 khiến việc mang đúng sản phẩm tới khách hàng trở nên thuận tiện hơn. Doanh thu, vì thế có thể được tăng lên.

17.1.2 Hai nhóm chính của recommendation system

Các recommendation system thường được chia thành hai nhóm lớn:

1. *Content-based system*: khuyến nghị dựa trên đặc tính của sản phẩm. Ví dụ, một người dùng xem rất nhiều các bộ phim về cảnh sát hình sự, vậy thì gợi ý một bộ phim trong cơ sở dữ liệu có chung đặc tính *hình sự* tới người dùng này, ví dụ phim *Người phán xử*. Cách tiếp cận này yêu cầu việc sắp xếp các sản phẩm vào từng nhóm hoặc đi tìm các đặc trưng của từng sản phẩm. Tuy nhiên, có những sản phẩm không có nhóm cụ thể và việc xác định nhóm hoặc đặc trưng của từng sản phẩm đôi khi là bất khả thi.
2. *Collaborative filtering*: hệ thống khuyến nghị các sản phẩm dựa trên sự tương quan (similarity) giữa các người dùng và/hoặc sản phẩm. Có thể hiểu rằng ở nhóm này một sản phẩm được *khuyến nghị* tới một người dùng dựa trên những người dùng có *hành vi* tương tự. Ví dụ, ba người dùng *A*, *B*, *C* đều thích các bài hát của Noo Phước Thịnh. Ngoài ra, hệ thống biết rằng người dùng *B*, *C* cũng thích các bài hát của Bích Phương nhưng chưa có thông tin về việc liệu người dùng *A* có thích Bích Phương hay không. Dựa trên thông tin của những người dùng tương tự là *B* và *C*, hệ thống có thể dự đoán rằng *A* cũng thích Bích Phương và gợi ý các bài hát của ca sĩ này tới *A*.

Trong chương này, chúng ta sẽ làm quen với nhóm thứ nhất, *content-based system*. Nhóm thứ hai, *collaborative filtering*, sẽ được thảo luận trong các chương còn lại của chương.

17.2 Utility matrix

Như đã đề cập, có hai thực thể chính trong các recommendation system là *user* và *item*. Mỗi *user* sẽ có *mức độ quan tâm* (*degree of preference*) tới từng *item* khác nhau. Mức độ quan tâm này, *nếu đã biết trước*, được gán cho một giá trị ứng với mỗi cặp *user-item*. Thông tin về mức độ quan tâm của một *user* tới một *item* có thể được thu thập thông qua một hệ

	A	B	C	D	E	F
Mưa nửa đêm	5	5	0	0	1	?
Cỏ úa	5	?	?	0	?	?
Vùng lá me bay	?	4	1	?	?	1
Con cò bé bé	1	1	4	4	4	?
Em yêu trường em	1	0	5	?	?	?

Hình 17.1: Ví dụ về utility matrix với hệ thống khuyến nghị bài hát. Các bài hát (*item*) được người dùng (*user*) đánh giá theo mức độ từ 0 đến 5 sao. Các dấu '?' nền màu xám ứng với việc dữ liệu chưa tồn tại trong cơ sở dữ liệu. Recommendation system cần phải *tự điền* các giá trị này.

thống đánh giá (*review* và *rating*); hoặc có thể dựa trên việc *user* đã click vào thông tin của *item* trên website; hoặc có thể dựa trên việc thời gian và số lần một *user* xem thông tin của một *item*. Các ví dụ trong phần này đều dựa trên hệ thống *rating*.

17.2.1 Ví dụ về utility matrix

Với một hệ thống *rating*, *mức độ quan tâm* của một *user* tới một *item* được đo bằng giá trị *user* đó đã đánh giá cho *item* đó, chẳng hạn số sao trên tổng cộng năm sao. Tập hợp tất cả các *rating*, bao gồm cả những giá trị chưa biết cần được dự đoán, tạo nên một ma trận gọi là ma trận *utility*. Xét ví dụ như trong Hình 17.1. Trong ví dụ này, có sáu *user* A, B, C, D, E, F và năm bài hát. Các ô màu xanh thể hiện việc một *user* đã đánh giá một bài hát với *rating* từ 0 (không thích) đến 5 (rất thích). Các ô có dấu '?' màu xám tương ứng với các ô chưa có dữ liệu. Công việc của một recommendation system là dự đoán giá trị tại các ô màu xám này, từ đó đưa ra gợi ý cho *user*. Vì vậy, bài toán recommendation system đôi khi được coi là bài toán *hoàn thiện ma trận* (*matrix completion*).

Trong ví dụ đơn giản này, dễ nhận thấy có hai thể loại nhạc khác nhau: ba bài đầu là nhạc *bolero* và hai bài sau là nhạc *thiếu nhi*. Từ dữ liệu này, ta cũng có thể đoán được rằng A, B thích thể loại nhạc *Bolero*; trong khi C, D, E, F thích thể loại nhạc *Thiếu nhi*. Từ đó, một hệ thống tốt nên gợi ý *Cỏ úa* cho B; *Vùng lá me bay* cho A; *Em yêu trường em* cho D, E, F. Giả sử chỉ có hai thể loại nhạc này, khi có một bài hát mới, ta chỉ cần phân lớp nó vào thể loại nào, từ đó đưa ra gợi ý với từng *user*.

Thông thường, có rất nhiều *user* và *item* trong hệ thống, và mỗi *user* thường chỉ *rate* một số lượng rất nhỏ các *item*, thậm chí có những *user* không *rate* *item* nào. Vì vậy, lượng ô màu xám của ma trận utility thường là rất lớn, và lượng các ô đã được điền là một số rất nhỏ.

Rõ ràng rằng càng nhiều ô được điền thì độ chính xác của hệ thống sẽ càng được cải thiện. Vì vậy, các hệ thống luôn khuyến khích *user bày tỏ* sự quan tâm của họ tới các *item* thông qua việc đánh giá các *item* đó. Việc đánh giá các *item*, vì thế, không những giúp các *user* khác biết được chất lượng của *item* đó mà còn giúp hệ thống *biết* được sở thích của *user*, qua đó có chính sách quảng cáo hợp lý.

17.2.2 Xây dựng ma trận utility

Không có ma trận utility, hệ thống gần như không thể gợi ý được *item* tới *user*, ngoài cách luôn luôn gợi ý các *item* phổ biến nhất. Vì vậy, trong các recommender system, việc xây dựng ma trận utility là tối quan trọng. Tuy nhiên, việc xây dựng ma trận này thường có gặp nhiều khó khăn. Có hai hướng tiếp cận phổ biến để xác định giá trị *rating* cho mỗi cặp *user-item* trong utility matrix:

1. Nhờ *user* đánh giá *item*. Amazon luôn nhờ *user* đánh giá các *item* của họ bằng cách gửi các email nhắc nhở nhiều lần. Tuy nhiên, cách tiếp cận này cũng có một vài hạn chế, vì thường thì *user* ít khi đánh giá sản phẩm. Và nếu có, đó có thể là những đánh giá thiên lệch bởi những người sẵn sàng đánh giá.
2. Hướng tiếp cận thứ hai là dựa trên hành vi của *user*. Nếu một *user* mua một *item* trên Amazon, xem một clip trên Youtube (có thể là nhiều lần), hay đọc một bài báo, có thể khẳng định *user* đó có xu hướng thích *item* đó. Facebook cũng dựa trên việc bạn *like* những nội dung nào để hiển thị trên *newsfeed* của bạn những nội dung liên quan. Bạn càng đam mê Facebook, Facebook càng được hưởng lợi, thế nên nó luôn mang tới bạn những thông tin mà khả năng cao là bạn *muốn* đọc. Thường thì với cách này, ta chỉ xây dựng được một ma trận với các thành phần là **1** và **0**, với **1** thể hiện *user* thích *item*, **0** thể hiện chưa có thông tin. Trong trường hợp này, **0** không có nghĩa là thấp hơn **1**, nó chỉ có nghĩa là *user* chưa cung cấp thông tin. Chúng ta cũng có thể xây dựng ma trận với các giá trị cao hơn 1 thông qua thời gian hoặc số lượt mà *user* xem một *item* nào đó. Ngoài ra, đôi khi nút *dislike* cũng mang lại những lợi ích nhất định cho hệ thống, lúc này có thể gán giá trị tương ứng bằng **-1**.

17.3 Content-based recommendation

17.3.1 Xây dựng *item profile*

Trong các hệ thống *content-based*, tức dựa trên *nội dung* của mỗi *item*, chúng ta cần xây dựng một *bộ hồ sơ* (*profile*) cho mỗi *item*. *Profile* này được biểu diễn dưới dạng toán học là một vector đặc trưng. Trong những trường hợp đơn giản, vector này được trực tiếp trích xuất từ *item*. Ví dụ, xem xét các thông tin của một bài hát mà có thể được sử dụng trong các recommendation system:

1. *Ca sĩ*. Cùng là bài *Thành phố buồn* nhưng có người thích bản của Đan Nguyên, có người lại thích bản của Đàm Vĩnh Hưng.
2. *Nhạc sĩ sáng tác*. Cùng là nhạc trẻ nhưng có người thích Phan Mạnh Quỳnh, người khác lại thích MTP.
3. *Năm sáng tác*. Một số người thích nhạc xưa cũ hơn nhạc hiện đại.
4. *Thể loại*. Quan họ và Bolero sẽ có thể thu hút những nhóm người khác nhau.

	A	B	C	D	E	F	item's feature vectors
Mưa nửa đêm	5	5	0	0	1	?	$\mathbf{x}_1 = [0.99, 0.02]^T$
Cỏ úa	5	?	?	0	?	?	$\mathbf{x}_2 = [0.91, 0.11]^T$
Vùng lá me bay	?	4	1	?	?	1	$\mathbf{x}_3 = [0.95, 0.05]^T$
Con cò bé bé	1	1	4	4	4	?	$\mathbf{x}_4 = [0.01, 0.99]^T$
Em yêu trường em	1	0	5	?	?	?	$\mathbf{x}_5 = [0.03, 0.98]^T$
User's models	θ_1	θ_2	θ_3	θ_4	θ_5	θ_6	← need to optimize

Hình 17.2: Giả sử feature vector cho mỗi sản phẩm được cho trong cột cuối cùng. Với mỗi người dùng, chúng ta cần tìm một mô hình θ_i tương ứng sao cho mô hình thu được là tốt nhất.

Có rất nhiều đặc trưng khác của một bài hát có thể được sử dụng. Ngoại trừ *Thể loại* khó định nghĩa, các yếu tố khác đều có thể được xác định rõ ràng.

Trong ví dụ ở Hình 17.1, chúng ta đơn giản hoá bài toán bằng việc xây dựng một vector đặc trưng hai chiều cho mỗi bài hát: chiều thứ nhất là mức độ *Bolero*, chiều thứ hai là mức độ *Thiếu nhi* của bài đó. Gọi các vector đặc trưng cho mỗi bài hát là $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5$. Giả sử các vector đặc trưng (ở dạng cột) cho mỗi bài hát được cho trong Hình 17.2. Ở đây, chúng ta tạm coi các vector này đã được xác định bằng một cách nào đó.

Tương tự như thế, hành vi của mỗi *user* cũng có thể được mô hình hoá dưới dạng tập các tham số θ . Dữ liệu huấn luyện để xây dựng mỗi mô hình θ_u là các cặp (*item profile, rating*) tương ứng với các *item* mà *user* đó đã đánh giá. Việc điền các giá trị còn thiếu trong ma trận utility chính là việc dự đoán mức độ quan tâm khi áp dụng mô hình θ_u lên chúng. Đầu ra này có thể được viết dưới dạng một hàm $f(\theta_u, \mathbf{x}_i)$. Việc lựa chọn dạng của $f(\theta_u, \mathbf{x}_i)$ tùy thuộc vào mỗi bài toán. Trong chương này, chúng ta sẽ quan tâm tới dạng đơn giản nhất—dạng tuyến tính.

17.3.2 Xây dựng hàm mất mát

Giả sử rằng số lượng *user* là N , số lượng *item* là M . Ma trận *profile* $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M] \in \mathbb{R}^{d \times M}$, và ma trận utility là $\mathbf{Y} \in \mathbb{R}^{M \times N}$. Thành phần ở hàng thứ m , cột thứ n của \mathbf{Y} là *mức độ quan tâm* (ở đây là số sao đã *rate*) của *user* thứ n lên *item* thứ m mà hệ thống đã thu thập được. Ma trận \mathbf{Y} bị khuyết rất nhiều thành phần tương ứng với các giá trị mà hệ thống cần dự đoán. Thêm nữa, gọi \mathbf{R} là ma trận *rated or not* thể hiện việc một *user* đã đánh giá một *item* hay chưa. Cụ thể, r_{mn} bằng 1 nếu *item* thứ m đã được đánh giá bởi *user* thứ n , bằng 0 trong trường hợp ngược lại.

Mô hình tuyến tính

Giả sử rằng ta có thể tìm được một mô hình cho mỗi *user*, được minh họa bởi một vector cột hệ số $\mathbf{w}_n \in \mathbb{R}^d$ và bias b_n sao cho *mức độ quan tâm* của một *user* tới một *item* có thể tính được bằng một hàm tuyến tính:

$$y_{mn} = \mathbf{w}_n^T \mathbf{x}_m + b_n \quad (17.1)$$

Xét một *user* thứ n bất kỳ, nếu ta coi tập huấn luyện là tập hợp các thành phần đã được *điền* của \mathbf{y}_n (cột thứ n của ma trận \mathbf{Y}), ta có thể xây dựng hàm mất mát tương tự như *ridge regression* (linear regression với l_2 regularization) như sau:

$$\mathcal{L}_n(\mathbf{w}_n, b_n) = \frac{1}{2s_n} \sum_{m:r_{mn}=1} (\mathbf{w}_n^T \mathbf{x}_m + b_n - y_{mn})^2 + \frac{\lambda}{2s_n} \|\mathbf{w}_n\|_2^2 \quad (17.2)$$

trong đó, thành phần thứ hai là regularization và λ là một tham số dương; s_n là số lượng các *item* mà *user* thứ n đã đánh giá, là tổng các phần tử trên cột thứ n của ma trận \mathbf{R} , tức $s_n = \sum_{m=1}^M r_{mn}$. Chú ý rằng regularization thường không được áp dụng lên bias b_n .

Vì biểu thức hàm mất mát (17.2) chỉ phụ thuộc vào các *item* đã được đánh giá, ta có thể rút gọn nó bằng cách đặt $\hat{\mathbf{y}}_n \in \mathbb{R}^{s_n}$ là *vector con* của \mathbf{y}_n , được xây dựng bằng cách trích các thành phần khác dấu '?' ở cột thứ n của \mathbf{Y} . Đồng thời, đặt $\hat{\mathbf{X}}_n \in \mathbb{R}^{d \times s_n}$ là *ma trận con* của ma trận đặc trưng \mathbf{X} , được tạo bằng cách trích các cột tương ứng với các *item* đã được đánh giá bởi *user* thứ n . (Xem ví dụ phía dưới để hiểu rõ hơn). Khi đó, biểu thức hàm mất mát của mô hình cho *user* thứ n được viết gọn thành:

$$\mathcal{L}_n(\mathbf{w}_n, b_n) = \frac{1}{2s_n} \|\hat{\mathbf{X}}_n^T \mathbf{w}_n + b_n \mathbf{e}_n - \hat{\mathbf{y}}_n\|_2^2 + \frac{\lambda}{2s_n} \|\mathbf{w}_n\|_2^2 \quad (17.3)$$

trong đó, \mathbf{e}_n là vector cột với tất cả các thành phần là 1. Đây chính xác là hàm mất mát của ridge regression. Cặp nghiệm \mathbf{w}_n, b_n có thể được tìm thông qua các thuật toán gradient descent. Trong chương này, chúng ta sẽ trực tiếp sử dụng class `Ridge` trong `sklearn.linear_model`. Có một điểm đáng lưu ý ở đây là \mathbf{w}_n chỉ được xác định nếu *user* thứ n đã đánh giá ít nhất một sản phẩm.

Chúng ta cùng theo dõi ví dụ nhỏ sau đây.

17.3.3 Ví dụ về hàm mất mát cho user E

Quay trở lại với ví dụ trong Hình 17.2, ma trận đặc trưng cho các *item* (mỗi cột tương ứng với một *item*) là

$$\mathbf{X} = \begin{bmatrix} 0.99 & 0.91 & 0.95 & 0.01 & 0.03 \\ 0.02 & 0.11 & 0.05 & 0.99 & 0.98 \end{bmatrix} \quad (17.4)$$

Xét trường hợp của *user* E với $n = 5$, $\mathbf{y}_5 = [1, ?, ?, 4, ?]^T \Rightarrow \mathbf{r}_5 = [1, 0, 0, 1, 0]^T$. Vì E mới chỉ đánh giá *item* thứ nhất và thứ tư nên $s_5 = 2$. Hơn nữa,

$$\hat{\mathbf{X}}_5 = \begin{bmatrix} 0.99 & 0.01 \\ 0.02 & 0.99 \end{bmatrix}, \hat{\mathbf{y}}_5 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, \mathbf{e}_5 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad (17.5)$$

Trong mỗi dòng, chúng ta sẽ thấy *id* của phim, tên phim, ngày phát hành, link trên imdb, và các số nhị phân **0, 1** phía cuối để chỉ ra bộ phim thuộc các thể loại nào trong 19 thể loại đã cho trong **u.genre**. Một bộ phim có thể thuộc nhiều thể loại khác nhau. Thông tin về thể loại này sẽ được dùng để xây dựng item profiles.

Với cơ sở dữ liệu này, chúng ta sẽ sử dụng thêm thư viện pandas (<http://pandas.pydata.org>) để đọc dữ liệu.

```
from __future__ import print_function
import numpy as np
import pandas as pd
# Reading user file:
u_cols = ['user_id', 'age', 'sex', 'occupation', 'zip_code']
users = pd.read_csv('ml-100k/u.user', sep='|', names=u_cols)
n_users = users.shape[0]
print('Number of users:', n_users)

#Reading ratings file:
r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']

ratings_base = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols)
ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols)

rate_train = ratings_base.as_matrix()
rate_test = ratings_test.as_matrix()

print('Number of traing rates:', rate_train.shape[0])
print('Number of test rates:', rate_test.shape[0])
```

Kết quả:

```
Number of users: 943
Number of traing rates: 90570
Number of test rates: 9430
```

Vì ta đang dựa trên thể loại của phim để xây dựng profile, ta sẽ chỉ quan tâm tới 19 giá trị nhị phân ở cuối mỗi hàng:

```
X0 = items.as_matrix()
X_train_counts = X0[:, -19:]
```

17.4.2 Xây dựng item profiles

Công việc quan trọng trong content-based recommendation system là xây dựng profile cho mỗi item, tức vector đặc trưng cho mỗi *item*. Trước hết, chúng ta cần load toàn bộ thông tin về các *item* vào biến **items**:

```
#Reading items file:
i_cols = ['movie id', 'movie title', 'release date', 'video release date', 'IMDb URL',
          'unknown', 'Action', 'Adventure', 'Animation', 'Children's', 'Comedy', 'Crime',
          'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical', 'Mystery',
          'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western']

items = pd.read_csv('ml-100k/u.item', sep='|', names=i_cols)

n_items = items.shape[0]
print('Number of items:', n_items)
```

Kết quả:

```
Number of items: 1682
```

Tiếp theo, chúng ta hiển thị một số hàng đầu tiên của ma trận `rate_train`

```
print(rate_train[:4, :])
```

Kết quả:

```
[[      1      1      5 874965758]
 [      1      2      3 876893171]
 [      1      3      4 878542960]
 [      1      4      3 876893119]]
```

Hàng thứ nhất được hiểu là *user* thứ nhất đánh giá *movie* thứ nhất 5 sao. Cột cuối cùng là một số chỉ thời điểm đánh giá, chúng ta sẽ bỏ qua thông số này.

Tiếp theo, chúng ta sẽ xây dựng feature vector cho mỗi item dựa trên ma trận thể loại phim và feature TF-IDF (<https://goo.gl/bpDdQ8>) trong thư viện `sklearn`.

```
#tfidf
from sklearn.feature_extraction.text import TfidfTransformer
transformer = TfidfTransformer(smooth_idf=True, norm='l2')
X = transformer.fit_transform(X_train_counts.tolist()).toarray()
```

Sau bước này, mỗi hàng của **X** tương ứng với vector đặc trưng của một bộ phim.

17.4.3 Tìm mô hình cho mỗi user

Với mỗi người dùng, chúng ta cần đi tìm những bộ phim nào mà người dùng đó đã đánh giá, và giá trị của các *rating* đó.

```
def get_itemsRated_by_user(rate_matrix, user_id):
    """
    return (item_ids, scores)
    """
    y = rate_matrix[:,0] # all users
    # item indices rated by user_id
    # we need to +1 to user_id since in the rate_matrix, id starts from 1
    # but id in python starts from 0
    ids = np.where(y == user_id + 1)[0]
    item_ids = rate_matrix[ids, 1] - 1 # index starts from 0
    scores = rate_matrix[ids, 2]
    return (item_ids, scores)
```

Bây giờ, ta có thể đi tìm các hệ số của Ridge Regression cho mỗi người dùng:

```
from sklearn.linear_model import Ridge
from sklearn import linear_model
d = X.shape[1] # data dimension
W = np.zeros((d, n_users))
b = np.zeros(n_users)
for n in range(n_users):
    ids, scores = get_itemsRated_by_user(rate_train, n)
    model = Ridge(alpha=0.01, fit_intercept = True)
    Xhat = X[ids, :]
    model.fit(Xhat, scores)
    W[:, n] = model.coef_
    b[n] = model.intercept_
```

Sau khi tính được các hệ số W và b , *rating* mà mỗi người dùng đánh giá mỗi bộ phim được dự đoán bằng cách:

```
# predicted scores
Yhat = X.dot(W) + b
```

Dưới đây là một ví dụ với người dùng có *id* là **10**.

```
n = 10
np.set_printoptions(precision=2) # 2 digits after .
ids, scores = get_itemsRated_by_user(rate_test, n)
print('Rated movies ids :', ids)
print('True ratings      :', scores)
print('Predicted ratings:', Yhat[ids, n])
```

Kết quả:

```
Rated movies ids : [ 37 109 110 226 424 557 722 724 731 739]
True ratings      : [3 3 4 3 4 3 5 3 3 4]
Predicted ratings: [3.18 3.13 3.42 3.09 3.35 5.2  4.01 3.35 3.42 3.72]
```

17.4.4 Đánh giá mô hình

Để đánh giá mô hình tìm được, chúng ta sẽ sử dụng Root Mean Squared Error (RMSE), tức căn bậc hai của trung bình cộng bình phương của lỗi.

```
def evaluate(Yhat, rates, W, b):
    se = cnt = 0
    for n in xrange(n_users):
        ids, scores_truth = get_itemsRatedByUser(rates, n)
        scores_pred = Yhat[ids, n]
        e = scores_truth - scores_pred
        se += (e*e).sum(axis = 0)
        cnt += e.size
    return np.sqrt(se/cnt)

print('RMSE for training: %.2f' %evaluate(Yhat, rate_train, W, b))
print('RMSE for test      : %.2f' %evaluate(Yhat, rate_test, W, b))
```

Kết quả:

```
RMSE for training: 0.91
RMSE for test      : 1.27
```

Như vậy, với training set, sai số vào khoảng 0.91 (sao); với test set, sai số lớn hơn một chút, khoảng 1.27. Các kết quả này chưa thực sự tốt vì mô hình đã được đơn giản hoá quá nhiều. Kết quả tốt hơn có thể được thấy trong các chương tiếp theo về collaborative filtering.

17.5 Thảo luận

- Content-based recommendation system là phương pháp đơn giản nhất trong các hệ thống recommendation system. Đặc điểm của phương pháp này là việc xây dựng mô hình cho mỗi *user* không phụ thuộc vào các *user* khác.
- Việc xây dựng mô hình cho mỗi *user* có thể được coi như bài toán regression hoặc classification với dữ liệu huấn luyện là các cặp (*item profile, rating*) mà *user* đó đã đánh giá. *Item profile* không phụ thuộc vào *user* mà phụ thuộc vào các đặc điểm mô tả của *item* hoặc cũng có thể được xác định bằng cách yêu cầu người dùng gắn *tag*.
- Source code trong chương này có thể được tìm thấy tại <https://goo.gl/u9M3vb>.

Đọc thêm

1. *Recommendation Systems–Stanford InfoLab* (<https://goo.gl/P1pesC>).
2. *Recommendation systems–Machine Learning, Andrew Ng* (<https://goo.gl/jdFvej>).
3. *Content Based Recommendations–Stanford University* (<https://goo.gl/3wnbZ4>).

Neighborhood-based collaborative filtering

18.1 Giới thiệu

Trong content-based recommendation system, chúng ta đã làm quen với một hệ thống gợi ý *item* đơn giản dựa trên vector đặc trưng của mỗi *item*. Đặc điểm của content-based recommendation system là việc xây dựng mô hình cho mỗi *user* không phụ thuộc vào các *user* khác mà phụ thuộc vào *profile* của các *item*. Việc làm này có lợi thế là tiết kiệm bộ nhớ và thời gian tính toán. Cách làm này có hai nhược điểm cơ bản. *Thứ nhất*, khi xây dựng mô hình cho một *user*, các hệ thống content-based không tận dụng được thông tin từ các *user* khác. Những thông tin này thường rất hữu ích vì hành vi mua hàng của các *user* thường được nhóm thành một vài nhóm đơn giản. Nếu biết hành vi mua hàng của một vài *user* trong nhóm, hệ thống nên có khả năng *suy luận* ra hành vi của những *user* còn lại. *Thứ hai*, không phải lúc nào chúng ta cũng có thể xây dựng *profile* cho mỗi *item*.

Những nhược điểm này có thể được giải quyết bằng một kỹ thuật có tên là *collaborative filtering*¹ (CF) [SFHS07, ERK⁺11]. Trong chương này, chúng ta cùng làm quen với một phương pháp CF có tên là *neighborhood-based collaborative filtering (NBCF)*. Chương tiếp theo sẽ trình bày về một phương pháp CF khác có tên *matrix factorization collaborative filtering*. Khi chỉ nói *collaborative filtering*, ta sẽ ngầm hiểu rằng đó là *neighborhood-based collaborative filtering*.

Ý tưởng của NBCF là xác định *mức độ quan tâm* của một *user* tới một *item* dựa trên hành vi của các *user* khác *gần giống* với *user* này. Việc *gần giống nhau* giữa các *user* có thể được xác định thông qua *mức độ quan tâm* của các *user* này tới các *item* khác mà hệ thống đã biết. Ví dụ, *A*, *B* đều thích phim *Cảnh sát hình sự*, đều đã đánh giá bộ phim này 5 sao. Ta đã biết *A* cũng thích *Người phán xử*, vậy nhiều khả năng *B* cũng thích bộ phim này.

¹ Tiếng Việt có tài liệu dịch là *lọc cộng hưởng*.

Các bạn có thể đã hình dung ra, hai câu hỏi quan trọng nhất trong một hệ thống neighborhood-based collaborative filtering là

1. Làm thế nào xác định được *sự giống nhau* giữa hai *user*?
2. Khi đã xác định được các *user gần giống nhau* (*similar user*) rồi, làm thế nào dự đoán được *mức độ quan tâm* của một *user* lên một *item*?

Việc xác định mức độ quan tâm của mỗi *user* tới một *item* dựa trên mức độ quan tâm của *user* tương tự tới *item* đó còn được gọi là *user-user collaborative filtering*. Có một hướng tiếp cận khác được cho là làm việc hiệu quả hơn là *item-item collaborative filtering*. Trong hướng tiếp cận này, thay vì xác định sự giống nhau giữa các *user*, hệ thống sẽ xác định sự giống nhau giữa các *item*. Từ đó, hệ thống gợi ý những *item gần giống* với những *item* mà *user* đó có mức độ quan tâm cao.

Cấu trúc của chương như sau: Mục 18.2 trình bày *user-user collaborative filtering*. Mục 18.3 nêu một số hạn chế của *user-user collaborative filtering* và cách khắc phục bằng *item-item collaborative filtering*. Kết quả của hai phương pháp này được trình bày qua ví dụ trên cơ sở dữ liệu MovieLens 100k trong Mục 18.4. Mục 18.5 thảo luận các ưu nhược điểm của NBCF.

18.2 User-user collaborative filtering

18.2.1 Hàm xác định độ giống nhau

Công việc quan trọng nhất phải làm trước tiên trong user-user collaborative filtering là phải xác định được *sự giống nhau* (*similarity*) giữa hai *user*. Giả sử dữ liệu duy nhất chúng ta có là *utility matrix* \mathbf{Y} , vậy *sự giống nhau* cần được xác định dựa trên các cột tương ứng với hai *user* trong ma trận này. Xét ví dụ trong Hình 18.1.

Giả sử có các *user* từ u_0 đến u_6 và các *item* từ i_0 đến i_4 trong đó các số trong mỗi ô vuông thể hiện số sao mà mỗi *user* đã đánh giá *item* đó với giá trị cao hơn thể hiện *mức độ quan tâm* cao hơn. Các dấu hỏi chấm là các giá trị mà hệ thống cần phải đi tìm. Đặt *mức độ giống nhau* của hai *user* u_i, u_j là $\text{sim}(u_i, u_j)$. Quan sát đầu tiên có thể nhận thấy là u_0, u_1 thích i_0, i_1, i_2 và không thích i_3, i_4 cho lắm. Điều ngược lại xảy ra ở các *user* còn lại. Vì vậy, một hàm đo *sự giống nhau similarity function* tốt cần đảm bảo

$$\text{sim}(u_0, u_1) > \text{sim}(u_0, u_i), \quad \forall i > 1. \quad (18.1)$$

Để xác định *mức độ quan tâm* của u_0 lên i_2 , chúng ta nên dựa trên hành vi của u_1 lên *item* này. Rất may rằng u_1 đã thích i_2 nên hệ thống cần khuyến nghị i_2 tới u_0 .

Câu hỏi đặt ra là, hàm số *similarity* cần được xây dựng như thế nào? Để đo *similarity* giữa hai *user*, cách thường làm là xây dựng một vector đặc trưng cho mỗi *user* rồi áp dụng một hàm có khả năng đo *similarity* giữa hai vector đó. Chú ý rằng việc xây dựng vector đặc trưng này khác với việc xây dựng *item profile* như trong content-based recommendation systems. Các vector này được xây dựng trực tiếp dựa trên ma trận utility chứ không dùng thêm thông

	u_0	u_1	u_2	u_3	u_4	u_5	u_6
i_0	5	5	2	0	1	?	?
i_1	3	?	?	0	?	?	?
i_2	?	4	1	?	?	1	2
i_3	2	2	3	4	4	?	4
i_4	2	0	4	?	?	?	5

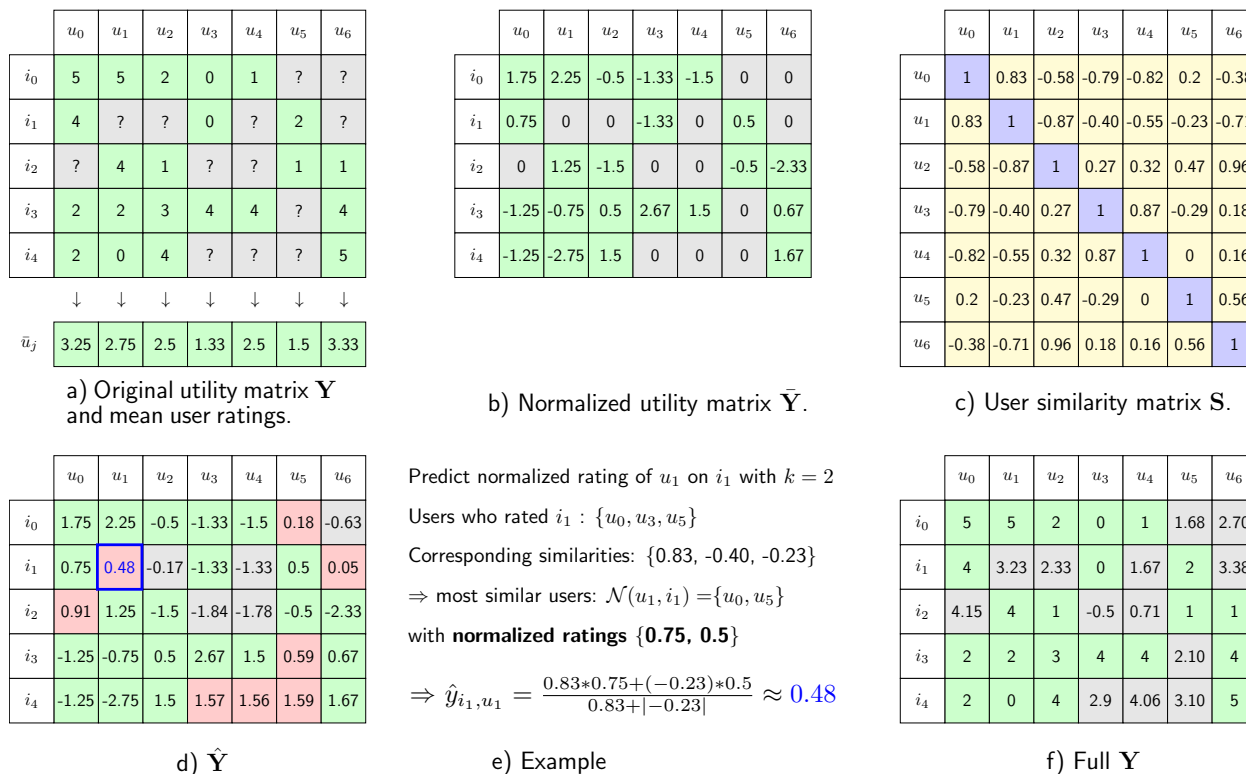
Hình 18.1: Ví dụ về utility matrix dựa trên số sao một *user* đánh giá một *item*. Một cách trực quan, hành vi của u_0 giống với u_1 hơn là u_2, u_3, u_4, u_5, u_6 . Từ đó có thể dự đoán rằng u_0 sẽ quan tâm tới i_2 vì u_1 cũng quan tâm tới *item* này.

tin bên ngoài như item profile. Với mỗi *user*, thông tin duy nhất chúng ta biết là các *rating* mà *user* đó đã thực hiện, tức cột tương ứng với *user* đó trong ma trận utility. Tuy nhiên, khó khăn là các cột này thường có rất nhiều giá trị bị khuyết (các dấu ‘?’ trong Hình 18.1) vì mỗi *user* thường chỉ đánh giá một số lượng rất nhỏ các *item*. Một cách khắc phục là giúp hệ thống ban đầu ước lượng thô các giá trị này sao cho việc điền không làm ảnh hưởng nhiều tới sự giống nhau giữa hai vector. Việc ước lượng này chỉ phục vụ cho việc tính *similarity*, không phải là kết quả cuối cùng hệ thống cần ước lượng.

Vậy mỗi dấu ‘?’ nên được thay bởi giá trị nào để hạn chế việc ước lượng bị sai lệch? Lựa chọn đầu tiên có thể nghĩ đến là thay các dấu ‘?’ bằng giá trị 0. Điều này không thực sự tốt vì giá trị 0 tương ứng với mức độ quan tâm thấp nhất; và một *user* chưa đánh giá một *item* không có nghĩa là họ hoàn toàn không quan tâm tới *item* đó. Một giá trị *an toàn* hơn là 2.5 vì nó là trung bình cộng của 0, mức thấp nhất, và 5, mức cao nhất. Tuy nhiên, giá trị này có hạn chế đối với những *user* dễ tính hoặc khó tính. Những *user* dễ tính có thể đánh giá ba sao cho các *item* họ không thích, ngược lại, những *user* khó tính có thể đánh giá ba sao cho những *item* họ thích. Việc thay đồng loạt các phần tử khuyết bởi 2.5 trong trường hợp này chưa mang lại hiệu quả. Một giá trị khả dĩ hơn cho việc này là ước lượng các phần tử khuyết như là giá trị trung bình mà một *user* đánh giá. Điều này giúp tránh việc một *user* quá khó tính hoặc dễ tính. Và các giá trị ước lượng này phụ thuộc vào từng *user*. Quan sát ví dụ trong Hình 18.2.

Hàng cuối cùng trong Hình 18.2a là trung bình của các đánh giá của mỗi *user*. Các giá trị cao tương ứng với các *user* dễ tính và ngược lại. Khi đó, nếu tiếp tục trừ từ mỗi *rating* đi giá trị trung bình này và thay các giá trị chưa biết bằng 0, ta sẽ được một *ma trận utility chuẩn hoá* (*normalized utility matrix*) như trong Hình 18.2b. Việc làm này có một vài ưu điểm:

- Việc trừ đi trung bình cộng của mỗi *cột* khiến mỗi *cột* có cả những giá trị dương và âm. Những *item* ứng với các giá trị dương có thể được coi như các *item* mà *user* đó quan tâm hơn so những *item* ứng với các giá trị âm. Những *item* mang giá trị bằng 0 chủ yếu ứng với việc *chưa xác định* được độ quan tâm của *user* đó.



Hình 18.2: Ví dụ mô tả User-user Collaborative Filtering. a) Utility Matrix ban đầu. b) Utility Matrix đã được chuẩn hoá. c) User similarity matrix. d) Dự đoán các (normalized) ratings còn thiếu. e) Ví dụ về cách dự đoán normalized rating của u_1 cho i_1 . f) Dự đoán các (denormalized) ratings còn thiếu.

- Về mặt kỹ thuật, số chiều của ma trận utility là rất lớn với hàng triệu *user* và *item*, nếu lưu toàn bộ các giá trị này trong một ma trận thì khả năng cao là sẽ không đủ bộ nhớ. Quan sát thấy rằng vì số lượng đánh giá biết trước thường là một số rất nhỏ so với kích thước của ma trận utility, sẽ tốt hơn nếu chúng ta lưu ma trận này dưới dạng một ma trận *sparse*, tức chỉ lưu các giá trị khác không và vị trí của chúng. Vì vậy, tốt hơn hết, các dấu '?' nên được thay bằng giá trị '0', tức chưa xác định liệu *user* có thích *item* hay không. Việc này không những tối ưu bộ nhớ mà việc tính toán ma trận *similarity* sau này cũng hiệu quả hơn. Ở đây, phần tử ở hàng thứ i , cột thứ j của ma trận *similarity* là độ *similarity* của *user* thứ i và thứ j .

Sau khi dữ liệu đã được chuẩn hoá, hàm *similarity* thường được sử dụng là *cosine similarity*:

$$\text{cosine_similarity}(\mathbf{u}_1, \mathbf{u}_2) = \cos(\mathbf{u}_1, \mathbf{u}_2) = \frac{\mathbf{u}_1^T \mathbf{u}_2}{\|\mathbf{u}_1\|_2 \cdot \|\mathbf{u}_2\|_2} \quad (18.2)$$

Trong đó $\mathbf{u}_{1,2}$ là các vector tương ứng với *user* 1 và *user* 2 như ở trên. Có một hàm trong Python phục vụ cách tính giá trị này một cách hiệu quả, chúng ta sẽ thấy trong phần lập trình.

Mức độ *similarity* của hai vector là một số thực trong đoạn $[-1, 1]$. Giá trị bằng 1 thể hiện hai vector hoàn toàn *similar* nhau. Hàm số \cos của một góc bằng 1 nghĩa là góc giữa hai vector bằng 0, tức hai vector có cùng phương và cùng hướng. Giá trị \cos bằng -1 thể hiện hai vector này hoàn toàn trái ngược nhau, tức cùng phương nhưng khác hướng. Điều này đồng nghĩa với việc nếu *hành vi* của hai *user* là hoàn toàn ngược nhau thì mức độ *similarity* giữa hai vector đó là thấp nhất.

Ví dụ về *cosine similarity* của các *user* (đã được chuẩn hoá) trong Hình 18.2b được cho trong Hình 18.2c. Ma trận *similarity* \mathbf{S} là một ma trận đối xứng vì \cos là một hàm chẵn², và nếu *user A giống user B* thì điều ngược lại cũng đúng. Các ô màu xanh trên đường chéo đều là \cos của góc giữa một vector và chính nó, tức $\cos(0) = 1$. Khi tính toán ở các bước sau, chúng ta không cần quan tâm tới các giá trị 1 này. Tiếp tục quan sát các vector hàng tương ứng với u_0, u_1, u_2 , chúng ta sẽ thấy một vài điều thú vị:

- u_0 gần với u_1 và u_5 (độ giống nhau là dương) hơn các *user* còn lại. Việc *similarity* cao giữa u_0 và u_1 là dễ hiểu vì cả hai đều có xu hướng quan tâm tới i_0, i_1, i_2 hơn các *item* còn lại. Việc u_0 gần với u_5 thoạt đầu có vẻ vô lý vì u_5 đánh giá thấp các *item* mà u_0 đánh giá cao (Hình 18.2a); tuy nhiên khi nhìn vào ma trận *utility* đã chuẩn hoá ở Hình 18.2b, ta thấy rằng điều này là hợp lý vì *item* duy nhất mà cả hai *user* này đã cung cấp thông tin là i_1 với các giá trị tương ứng đều là *tích cực*.
- u_1 gần với u_0 và xa các *user* còn lại.
- u_2 gần với u_3, u_4, u_5, u_6 và xa các *user* còn lại.

Từ ma trận *similarity* này, chúng ta có thể phân nhóm các *user* ra làm hai nhóm (u_0, u_1) và (u_2, u_3, u_4, u_5, u_6). Vì ma trận \mathbf{S} này nhỏ nên chúng ta có thể dễ dàng quan sát thấy điều này; khi số *user* lớn hơn, việc xác định bằng *mắt thường* là không khả thi. Việc xây dựng thuật toán phân nhóm các *user* (*users clustering*) sẽ được trình bày trong chương tiếp theo.

Có một chú ý quan trọng ở đây là khi số lượng *user* lớn, ma trận \mathbf{S} cũng rất lớn và nhiều khả năng là không có đủ bộ nhớ để lưu trữ, ngay cả khi chỉ lưu hơn một nửa số các phần tử của ma trận đối xứng này. Với các trường hợp đó, mới mỗi *user*, chúng ta chỉ cần tính và lưu kết quả của một hàng của *similarity matrix*, tương ứng với việc độ *giống nhau* giữa *user* đó và các *user* còn lại.

18.2.2 Điền các giá trị khuyết trong ma trận *utility*

Việc *dự đoán mức độ quan tâm (predicted rating)* của một *user* lên một *item* dựa trên các *user gần nhất* này rất giống với những gì chúng ta thấy trong *K-nearest neighbors* (KNN) với hàm khoảng cách là *cosine similarity*.

Tương tự như KNN, NBCF cũng dùng thông tin của k *user* lân cận để dự đoán. Tất nhiên, để đánh giá độ quan tâm của một *user* lên một *item*, chúng ta chỉ quan tâm tới các *user*

² Một hàm số $f: \mathbb{R} \rightarrow \mathbb{R}$ được gọi là *chẵn* nếu $f(x) = f(-x)$, $\forall x \in \mathbb{R}$.

trong lân cận **đã đánh giá** *item* đó. *Giá trị cần điền* thường được xác định là *trung bình có trọng số* của các *rating đã chuẩn hoá*. Có một điểm cần lưu ý, trong KNN, các trọng số được xác định dựa trên khoảng cách giữa hai điểm, và các khoảng cách này là các số không âm. Trong NBCF, các trọng số được xác định dựa trên *similarity* giữa hai *user*, những trọng số này có thể nhỏ hơn 0. Công thức phổ biến được sử dụng để dự đoán số sao mà *user u* đánh giá *item i* là³

$$\hat{y}_{i,u} = \frac{\sum_{u_j \in \mathcal{N}(u,i)} \bar{y}_{i,u_j} \text{sim}(u, u_j)}{\sum_{u_j \in \mathcal{N}(u,i)} |\text{sim}(u, u_j)|} \quad (18.3)$$

trong đó $\mathcal{N}(u, i)$ là tập hợp k *user* gần giống nhất, tức có *similarity* cao nhất của u đã đánh giá i . Hình 18.2d thể hiện việc điền các giá trị còn thiếu trong ma trận *utility* đã chuẩn hoá. Các ô màu nền đỏ thể hiện các giá trị dương, tức các *item* mà có thể *user* đó *quan tâm*. Ở đây, ngưỡng được lấy là 0, ngưỡng này hoàn toàn có thể được thay đổi tùy thuộc vào việc ta muốn gợi ý nhiều hay ít *item*.

Một ví dụ về việc tính *normalized rating* của u_1 cho i_1 được cho trong Hình 18.2e với số *nearest neighbors* là $k = 2$. Các bước thực hiện như sau

1. Xác định các *user* đã đánh giá i_1 , chúng là u_0, u_3, u_5 .
2. Mức độ *similarity* của u_1 với các *user* này lần lượt là $\{0.83, -0.40, -0.23\}$. Hai ($k = 2$) giá trị lớn nhất là 0.83 và -0.23 tương ứng với u_0 và u_5 .
3. Xác định các đánh giá (đã chuẩn hoá) của u_0 và u_5 cho i_1 , ta thu được hai giá trị lần lượt là 0.75 và 0.5.
4. Dự đoán kết quả

$$\hat{y}_{i_1, u_1} = \frac{0.83 \times 0.75 + (-0.23) \times 0.5}{0.83 + |-0.23|} \approx 0.48 \quad (18.4)$$

Việc quy đổi các giá trị đánh giá đã chuẩn hoá về thang 5 có thể được thực hiện bằng cách cộng các cột của ma trận \hat{Y} với giá trị đánh giá trung bình của mỗi *user* như đã tính trong Hình 18.2a. Việc hệ thống quyết định gợi ý *item* nào cho mỗi *user* có thể được xác định bằng nhiều cách khác nhau. Hệ thống có thể sắp xếp các *item* chưa được đánh giá theo độ giảm dần của *predicted rating*, hoặc có thể chỉ chọn các *item* có *normalized predicted rating* dương–tương ứng với việc *user* này có nhiều khả năng thích hơn.

18.3 Item-item collaborative filtering

User-user CF có một số hạn chế như sau:

- Khi số lượng *user* lớn hơn số lượng *item* rất nhiều (điều này thường xảy ra), kích thước ma trận *similarity* là rất lớn (mỗi chiều của ma trận này có số phần tử chính bằng số *user*). Việc lưu trữ một ma trận với kích thước lớn nhiều khi không khả thi.

³ Sự khác biệt so với trung bình có trọng số là mẫu số có sử dụng trị tuyệt đối để xử lý các số âm.

- Ma trận utility \mathbf{Y} thường rất *sparse*, tức chỉ có một tỉ lệ nhỏ các phần tử đã biết. Với số lượng *user* rất lớn so với số lượng *item*, rất nhiều cột của ma trận này có rất ít, thậm chí không có phần tử khác 0 vì các *user* thường *lười* đánh giá *item*. Cũng chính vì thế, một khi *user* đó thay đổi các *rating* trước đó hoặc đánh giá thêm *item*, trung bình cộng các *rating* cũng như vector chuẩn hoá tương ứng với *user* này thay đổi nhiều. Kéo theo đó, việc tính toán ma trận similarity, vốn tốn nhiều bộ nhớ và thời gian, cũng cần được thực hiện lại.

Có một cách tiếp cận khác, thay vì tìm sự giống nhau giữa các *user*, ta có thể tìm sự giống nhau giữa các *item*. Từ đó nếu một *user* thích một *item* thì hệ thống nên gợi ý các *item* tương tự tới *user* đó. Việc này có một số ưu điểm:

- Khi số lượng *item* nhỏ hơn số lượng *user*, ma trận similarity có kích thước nhỏ hơn, việc này khiến việc lưu trữ và tính toán ở các bước sau được thực hiện một cách hiệu quả hơn.
- Cũng giả sử rằng số lượng *item* ít hơn số lượng *user*. Vì tổng lượng đánh giá là không đổi, số lượng trung bình các *item* được đánh giá bởi một *user* sẽ ít hơn số lượng trung bình các *user* đã đánh giá một *item*. Nói cách khác, nếu ma trận utility có số hàng ít hơn số cột, số lượng phần tử trung bình đã biết trong mỗi hàng sẽ nhiều hơn số lượng phần tử trung bình đã biết trong mỗi cột. Kéo theo đó, thông tin về mỗi *item* là nhiều hơn thông tin về mỗi *user*, việc tính độ *similarity* giữa các hàng cũng đáng tin cậy hơn. Hơn nữa, giá trị trung bình của mỗi hàng cũng thay đổi ít hơn khi có thêm một vài đánh giá. Như vậy, việc cập nhật ma trận similarity có thể được thực hiện ít thường xuyên hơn.

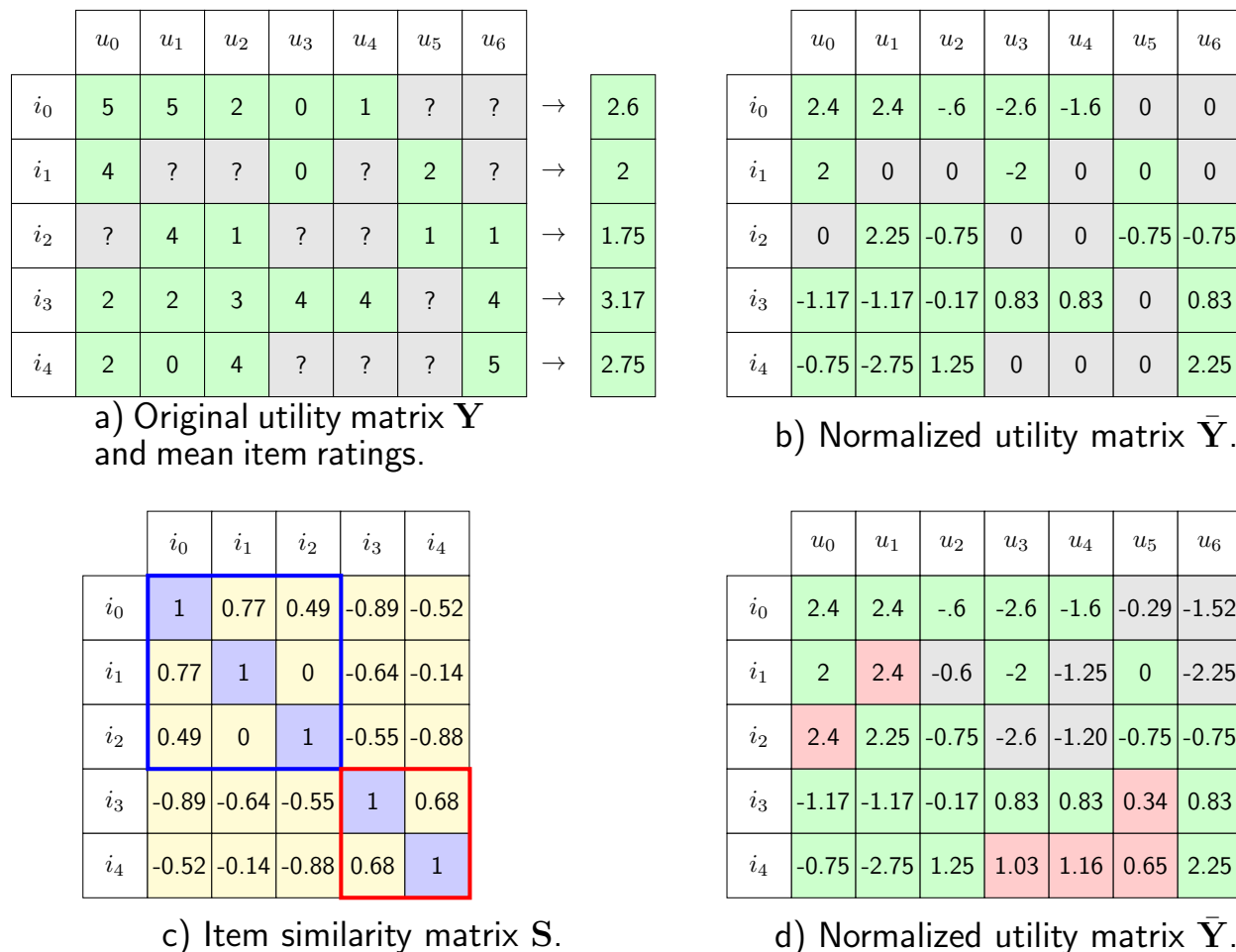
Cách tiếp cận thứ hai này được gọi là *item-item collaborative filtering* (item-item CF). Khi số lượng *item* ít hơn số lượng *user*, phương pháp này được ưu tiên sử dụng hơn.

Quy trình dự đoán các đánh giá bị khuyết cũng tương tự như trong user-user CF, chỉ khác là bây giờ ta cần tính độ *giống nhau* giữa các hàng.

Liên hệ giữa item-item CF và user-user CF

Về mặt tính toán, *item-item CF* có thể nhận được từ *user-user CF* bằng cách chuyển vị (transpose) ma trận utility, và coi như *item* đang đánh giá ngược *user*. Sau khi tính ra kết quả cuối cùng, ta lại chuyển vị một lần nữa để thu được kết quả.

Hình 18.3 mô tả quy trình này với ví dụ nêu ở phần trên. Có một điểm thú vị trong ma trận similarity ở Hình 18.3c là có các phần tử trong hai khu vực hình vuông xanh và đỏ đều là các số không âm, các phần tử bên ngoài là các số âm. Việc này thể hiện rằng các *item* có thể được chia thành hai nhóm rõ rệt với những *item* có *similarity* không âm trong một nhóm cột vào một nhóm. Như vậy, một cách *vô tình*, chúng ta đã thực hiện việc *item clustering*. Việc này sẽ giúp ích rất nhiều trong việc dự đoán ở phần sau vì các *item* gần giống nhau rất có thể đã được phân vào một nhóm. Kết quả cuối cùng về việc chọn *item* nào để *recommend* cho mỗi *user* được thể hiện bởi các ô màu đỏ trong Hình 18.3d. Kết quả



Hình 18.3: Ví dụ mô tả item-item CF. a) Ma trận utility ban đầu. b) Ma trận utility đã được chuẩn hoá. c) User similarity matrix. d) Dự đoán các (normalized) *rating* còn thiếu.

này có khác một chút so với kết quả tìm được bởi user-user CF ở hai cột cuối cùng tương ứng với u_5, u_6 . Dường như kết quả này *hợp lý* hơn vì từ utility matrix, ta nhận thấy có hai nhóm *user* thích hai nhóm *item* khác nhau. Nhóm thứ nhất là u_0 và u_1 ; nhóm thứ hai là các *user* còn lại.

Mục 18.4 sau đây mô tả cách lập trình cho NNCF trên Python. Chú ý rằng thư viện `sklearn` chưa hỗ trợ các module cho recommendation system. Một thư viện khác khá tốt trên python bạn đọc có thể tham khảo là `surprise` (<http://surpriselib.com/>).

18.4 Lập trình trên Python

Thuật toán collaborative filtering trong chương này tương đối đơn giản và không chứa bài toán tối ưu nào. Chúng ta tiếp tục sử dụng bộ cơ sở dữ liệu MovieLens 100k như trong chương trước. Dưới đây là đoạn code thể hiện `class uuCF` cho user-user collaborative filtering. Có hai phương thức chính của `class` này là `fit`—tính ma trận similarity, và `predict`—dự đoán số sao mà một *user* sẽ đánh giá một *item*.


```

from __future__ import print_function
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from scipy import sparse

class uuCF(object):
    def __init__(self, Y_data, k, sim_func = cosine_similarity):
        self.Y_data = Y_data # a 2d array of shape (n_users, 3)
            # each row of Y_data has form [user_id, item_id, rating]
        self.k = k # number of neighborhood
        self.sim_func = sim_func # similarity function, default: cosine_similarity
        self.Ybar = None # normalize data
        self.n_users = int(np.max(self.Y_data[:, 0])) + 1 # number of users
        self.n_items = int(np.max(self.Y_data[:, 1])) + 1 # number of items

    def fit(self):
        # normalized Y_data -> Ybar
        users = self.Y_data[:, 0] # all users - first column of Y_data
        self.Ybar = self.Y_data.copy()
        self.mu = np.zeros((self.n_users,))
        for n in xrange(self.n_users):
            # row indices of ratings made by user n
            ids = np.where(users == n)[0].astype(np.int32)
            # indices of all items rated by user n
            item_ids = self.Y_data[ids, 1]
            # ratings made by user n
            ratings = self.Y_data[ids, 2]
            # avoid zero division
            self.mu[n] = np.mean(ratings) if ids.size > 0 else 0
            self.Ybar[ids, 2] = ratings - self.mu[n]

        ## form the rating matrix as a sparse matrix.
        # see more: https://goo.gl/i2mmT2
        self.Ybar = sparse.coo_matrix((self.Ybar[:, 2],
            (self.Ybar[:, 1], self.Ybar[:, 0])), (self.n_items, self.n_users)).tocsr()
        self.S = self.sim_func(self.Ybar.T, self.Ybar.T)

    def pred(self, u, i):
        """ predict the rating of user u for item i"""
        # find item i
        ids = np.where(self.Y_data[:, 1] == i)[0].astype(np.int32)
        # all users who rated i
        users_rated_i = (self.Y_data[ids, 0]).astype(np.int32)
        # similarity of u and users who rated i
        sim = self.S[u, users_rated_i]
        # most k similar users
        nns = np.argsort(sim)[-self.k:]
        nearest_s = sim[nns] # and the corresponding similarities
        # the corresponding ratings
        r = self.Ybar[i, users_rated_i[nns]]
        eps = 1e-8 # a small number to avoid zero division
        return (r*nearest_s).sum()/(np.abs(nearest_s).sum() + eps) + self.mu[u]

```

Tiếp theo, ta áp dụng vào MoviesLen 100k:

```

r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings_base = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols)
ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols)

rate_train = ratings_base.as_matrix()
rate_test = ratings_test.as_matrix()

# indices start from 0
rate_train[:, :2] -= 1
rate_test[:, :2] -= 1

rs = uuCF(rate_train, k = 40)
rs.fit()

n_tests = rate_test.shape[0]
SE = 0 # squared error
for n in xrange(n_tests):
    pred = rs.pred(rate_test[n, 0], rate_test[n, 1])
    SE += (pred - rate_test[n, 2])**2

RMSE = np.sqrt(SE/n_tests)
print('User-user CF, RMSE =', RMSE)

```

Kết quả:

```
User-user CF, RMSE = 0.976614028929
```

Như vậy, trung bình mỗi *rating* bị dự đoán sai lệch khoảng 0.976. Kết quả này có tốt hơn kết quả có được bởi content-based recommendation system.

Tiếp theo, chúng ta áp dụng item-item CF vào tập cơ sở dữ liệu này. Để áp dụng item-item CF, chúng ta chỉ cần chuyển vị ma trận utility. Trong trường hợp này, vì ma trận utility được lưu dưới dạng `[user_id, item_id, rating]` nên ta chỉ cần đổi chỗ cột thứ nhất cho cột thứ hai của `Y_data`:

```

rate_train = rate_train[:, [1, 0, 2]]
rate_test = rate_test[:, [1, 0, 2]]

rs = uuCF(rate_train, k = 40)
rs.fit()

n_tests = rate_test.shape[0]
SE = 0 # squared error
for n in xrange(n_tests):
    pred = rs.pred(rate_test[n, 0], rate_test[n, 1])
    SE += (pred - rate_test[n, 2])**2

RMSE = np.sqrt(SE/n_tests)
print('Item-item CF, RMSE =', RMSE)

```

Kết quả:

```
Item-item CF, RMSE = 0.968846083868
```

Như vậy, trong trường hợp này item-item collaborative filtering cho kết quả tốt hơn, ngay cả khi số *item* (1682) lớn hơn số lượng *user* (943). Với các bài toán khác, chúng ta nên thử cả hai trên một tập validation và chọn ra phương pháp cho kết quả tốt hơn. Chúng ta cũng có thể thay *kích thước lân cận k* bằng các giá trị khác và so sánh các kết quả.

18.5 Thảo luận

- CF là một phương pháp gợi ý *item* với ý tưởng chính dựa trên hành vi của các *user* tương tự khác lên cùng một *item*. Việc suy ra này được thực hiện dựa trên ma trận *similarity* đo độ giống nhau giữa các *user*.
- Để tính ma trận *similarity*, trước tiên ta cần chuẩn hoá dữ liệu. Phương pháp phổ biến là *mean offset*, tức trừ các *ratings* đi giá trị trung bình mà một *user* đưa ra cho các *item*.
- Similarity function thường được dùng là **cosine similarity**.
- Một hướng tiếp cận tương tự là thay vì đi tìm các *user* gần giống với một *user* (user-user CF), ta đi tìm các *item* gần với một *item* cho trước (item-item CF). Trên thực tế, item-item CF thường cho kết quả tốt hơn.
- Source code của chương này có thể được tìm thấy tại <https://goo.gl/vGKjbo>.

Đọc thêm

1. M. Ekstrand *et al.*, *Collaborative filtering recommender systems*. (<https://goo.gl/GVn8av>) Foundations and Trends® in Human-Computer Interaction 4.2 (2011): 81-173.

Matrix factorization collaborative filtering

19.1 Giới thiệu

Trong Chương 18, chúng ta đã làm quen với một phương pháp collaborative filtering (CF) dựa trên hành vi của các *user* hoặc *item* lân cận. Trong chương này, chúng ta sẽ làm quen với một hướng tiếp cận khác cho collaborative filtering dựa trên bài toán *phân tích ma trận thành nhân tử* (*matrix factorization* hoặc *matrix decomposition*). Phương pháp này được gọi là *matrix factorization collaborative filtering* (MF-CF) [KBV09].

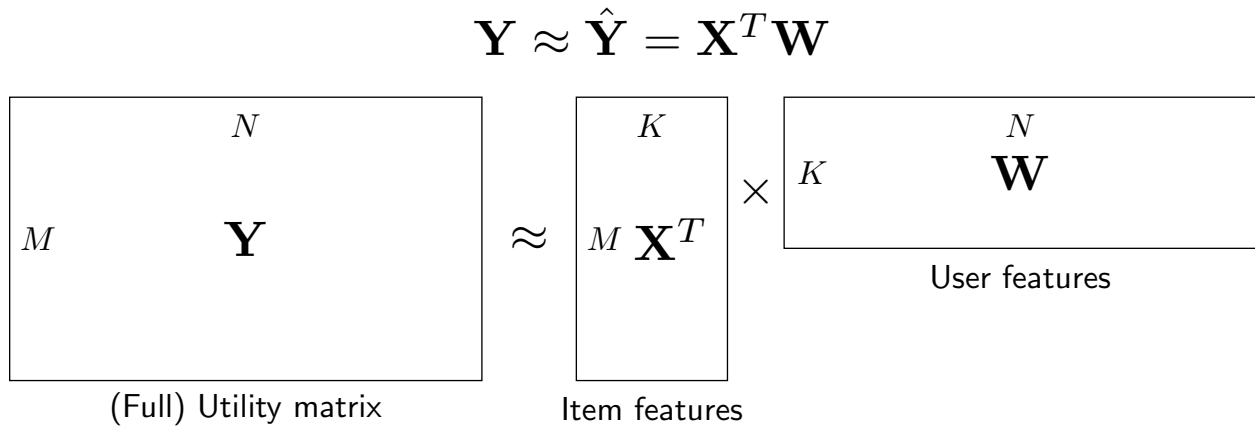
Nhắc lại rằng trong content-based recommendation systems, mỗi *item* được mô tả bằng một vector \mathbf{x} được gọi là *item profile*. Trong phương pháp đó, ta cần tìm một vector hệ số \mathbf{w} tương ứng với mỗi *user* sao cho *rating* đã biết mà *user* đó cho *item* xấp xỉ với

$$y \approx \mathbf{w}^T \mathbf{x} = \mathbf{x}^T \mathbf{w} \quad (19.1)$$

Với cách làm này, ma trận utility \mathbf{Y} , *giả sử đã được điền hết*, sẽ xấp xỉ với:

$$\mathbf{Y} \approx \begin{bmatrix} \mathbf{x}_1^T \mathbf{w}_1 & \mathbf{x}_1^T \mathbf{w}_2 & \dots & \mathbf{x}_1^T \mathbf{w}_N \\ \mathbf{x}_2^T \mathbf{w}_1 & \mathbf{x}_2^T \mathbf{w}_2 & \dots & \mathbf{x}_2^T \mathbf{w}_N \\ \dots & \dots & \ddots & \dots \\ \mathbf{x}_M^T \mathbf{w}_1 & \mathbf{x}_M^T \mathbf{w}_2 & \dots & \mathbf{x}_M^T \mathbf{w}_N \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \dots \\ \mathbf{x}_M^T \end{bmatrix} [\mathbf{w}_1 \ \mathbf{w}_2 \ \dots \ \mathbf{w}_N] = \mathbf{X}^T \mathbf{W} \quad (19.2)$$

với M, N lần lượt là số lượng *item* và *user*. Chú ý rằng trong content-based collaborative filtering, \mathbf{x} được xây dựng dựa trên thông tin mô tả của *item* và quá trình xây dựng này độc lập với quá trình đi tìm hệ số phù hợp cho mỗi *user*. Như vậy, việc xây dựng *item profile* đóng vai trò rất quan trọng và có ảnh hưởng trực tiếp lên hiệu năng của mô hình. Thêm nữa, việc xây dựng từng mô hình riêng lẻ cho mỗi *user* dẫn đến kết quả chưa thực sự tốt vì không khai thác được mối quan hệ giữa các *user*.



Hình 19.1: Matrix factorization. Ma trận utility $\mathbf{Y} \in \mathbb{R}^{M \times N}$ được phân tích thành tích của hai ma trận $\mathbf{X} \in \mathbb{R}^{M \times K}$ và $\mathbf{W} \in \mathbb{R}^{K \times N}$.

Bây giờ, giả sử rằng ta không cần xây dựng từ trước các *item profile* \mathbf{x} mà vector đặc trưng cho mỗi *item* này có thể được *huấn luyện* đồng thời với mô hình của mỗi *user* (ở đây là một vector hệ số). Điều này nghĩa là, biến số trong bài toán tối ưu là cả \mathbf{X} và \mathbf{W} ; trong đó, \mathbf{X} là ma trận của toàn bộ *item profile*, mỗi **cột** tương ứng với một *item*, \mathbf{W} là ma trận của toàn bộ *user model*, mỗi **cột** tương ứng với một *user*.

Với cách làm này, chúng ta đang cố gắng xấp xỉ ma trận utility $\mathbf{Y} \in \mathbb{R}^{M \times N}$ bằng tích của hai ma trận $\mathbf{X} \in \mathbb{R}^{K \times M}$ và $\mathbf{W} \in \mathbb{R}^{K \times N}$. Thông thường, K được chọn là một số nhỏ hơn rất nhiều so với M, N . Khi đó, cả hai ma trận \mathbf{X} và \mathbf{W} đều có rank không vượt quá K . Chính vì vậy, phương pháp này còn được gọi là *low-rank matrix factorization* (xem Hình 19.1).

Có một vài điểm cần lưu ý:

- Ý tưởng chính đằng sau matrix factorization cho recommendation system là tồn tại các *đặc trưng ẩn* (*latent feature*) mô tả sự liên quan giữa các *item* và các *user*. Ví dụ, trong hệ thống khuyến nghị các bộ phim, tính chất ẩn có thể là *hành sự*, *chính trị*, *hành động*, *hài*, v.v.; cũng có thể là một sự kết hợp nào đó của các thể loại này; hoặc cũng có thể là bất cứ điều gì mà chúng ta không thực sự cần đặt tên. Mỗi *item* sẽ mang tính chất ẩn ở một mức độ nào đó tương ứng với các hệ số trong vector \mathbf{x} của nó, hệ số càng cao tương ứng với việc mang tính chất đó càng cao. Tương tự, mỗi *user* cũng sẽ có xu hướng thích những tính chất ẩn nào đó và được mô tả bởi các hệ số trong vector \mathbf{w} của nó. Hệ số cao tương ứng với việc *user* thích các bộ phim có tính chất ẩn đó. Giá trị của biểu thức $\mathbf{x}^T \mathbf{w}$ sẽ cao nếu các thành phần tương ứng của \mathbf{x} và \mathbf{w} đều cao (và dương). Điều này nghĩa là *item* mang các tính chất ẩn mà *user* thích, vậy ta nên gợi ý *item* này cho *user* đó.
- Tại sao matrix factorization lại được xếp vào collaborative filtering? Câu trả lời đến từ việc đi tối ưu hàm mất mát mà chúng ta sẽ thảo luận ở Mục 19.2. Về cơ bản, để tìm nghiệm của bài toán tối ưu, ta phải lần lượt đi tìm \mathbf{X} và \mathbf{W} khi thành phần còn lại được cố định. Như vậy, mỗi cột của \mathbf{X} sẽ phụ thuộc vào toàn bộ các cột của \mathbf{W} . Ngược lại,

mỗi cột của \mathbf{W} lại phụ thuộc vào toàn bộ các cột của \mathbf{X} . Như vậy, có những mối quan hệ ràng buộc *chằng chịt* giữa các thành phần của hai ma trận trên. Tức chúng ta cần sử dụng thông tin của tất cả để suy ra tất cả. Vậy nên phương pháp này cũng được xếp vào collaborative filtering.

- Trong các bài toán thực tế, số lượng *item* M và số lượng *user* N thường rất lớn. Việc tìm ra các mô hình đơn giản giúp dự đoán các *rating* cần được thực hiện một cách nhanh nhất có thể. Neighborhood-based collaborative filtering không yêu cầu việc huấn luyện quá nhiều, nhưng trong quá trình dự đoán, ta cần đi tìm độ *similarity* của *user* đang xét với *tất cả* các *user* còn lại rồi suy ra kết quả. Ngược lại, với matrix factorization, việc huấn luyện có thể hơi phức tạp một chút vì phải lặp đi lặp lại việc tối ưu một ma trận khi cố định ma trận còn lại, nhưng việc dự đoán đơn giản hơn vì ta chỉ cần lấy tích vô hướng của hai vector $\mathbf{x}^T \mathbf{w}$, mỗi vector có độ dài K là một số nhỏ hơn nhiều so với M, N . Vì vậy, quá trình dự đoán không yêu cầu khả năng tính toán cao. Việc này khiến nó phù hợp với các mô hình có tập dữ liệu lớn.
- Thêm nữa, việc lưu trữ hai ma trận \mathbf{X} và \mathbf{W} yêu cầu lượng bộ nhớ nhỏ so với việc lưu toàn bộ ma trận utility và similarity trong neighborhood-based collaborative filtering. Cụ thể, ta cần bộ nhớ để chứa $K(M + N)$ phần tử thay vì M^2 hoặc N^2 của ma trận *similarity*.

19.2 Xây dựng và tối ưu hàm mất mát

19.2.1 Xấp xỉ các đánh giá đã biết

Như đã đề cập, đánh giá của *user* n lên *item* m có thể được xấp xỉ bởi $y_{mn} = \mathbf{x}_m^T \mathbf{w}_n$. Ta cũng có thể thêm các bias vào công thức xấp xỉ này và tối ưu các bias đó. Cụ thể:

$$y_{mn} \approx \mathbf{x}_m^T \mathbf{w}_n + b_m + d_n \quad (19.3)$$

Trong đó, b_m và d_n lượt lượt là các hệ số tự do tương ứng với *item* m và *user* n . Vector $\mathbf{b} = [b_1, b_2, \dots, b_M]^T$ là vector bias cho các *item*, vector $\mathbf{d} = [d_1, d_2, \dots, d_N]^T$ là vector bias cho các *user*. Giống như trong neighborhood-based collaborative filtering (NBCF), các giá trị này cũng có thể được coi là các giá trị giúp chuẩn hoá dữ liệu với \mathbf{b} tương ứng với item-item CF và \mathbf{d} tương ứng với user-user CF. Không giống như trong NBCF, các giá trị này sẽ được tối ưu để tìm ra các giá trị giúp xấp xỉ tập huấn luyện tốt nhất. Thêm vào đó, huấn luyện cùng lúc cả \mathbf{d} và \mathbf{b} giúp kết hợp cả user-user CF và item-item CF vào trong một bài toán tối ưu. Vì vậy, chúng ta mong đợi rằng phương pháp này sẽ mang lại hiệu quả tốt hơn.

19.2.2 Hàm mất mát

Hàm mất mát cho MF-CF có thể được viết như sau

$$\mathcal{L}(\mathbf{X}, \mathbf{W}, \mathbf{b}, \mathbf{d}) = \underbrace{\frac{1}{2s} \sum_{n=1}^N \sum_{m:r_{mn}=1} (\mathbf{x}_m^T \mathbf{w}_n + b_m + d_n - y_{mn})}_{\text{data loss}} + \underbrace{\frac{\lambda}{2} (\|\mathbf{X}\|_F^2 + \|\mathbf{W}\|_F^2)}_{\text{regularization loss}} \quad (19.4)$$

trong đó $r_{mn} = 1$ nếu *item* thứ m đã được đánh giá bởi *user* thứ n , s là số lượng *rating* trong tập huấn luyện, y_{mn} là *rating* chưa chuẩn hoá¹ của *user* thứ n cho *item* thứ m . Thành phần thứ nhất của hàm mất mát, *data loss*, chính là trung bình sai số của mô hình. Thành phần thứ hai, *regularization loss*, là l_2 regularization, thành phần này giúp tránh overfitting².

Việc tối ưu đồng thời $\mathbf{X}, \mathbf{W}, \mathbf{b}, \mathbf{d}$ là tương đối phức tạp. Thay vào đó, phương pháp được sử dụng là lần lượt tối ưu một trong hai cặp $(\mathbf{X}, \mathbf{b}), (\mathbf{W}, \mathbf{d})$ khi cố định cặp còn lại. Quá trình này được lặp đi lặp lại tới khi hàm mất mát hội tụ.

19.2.3 Tối ưu hàm mất mát

Khi cố định cặp (\mathbf{X}, \mathbf{b}) , bài toán tối ưu cặp (\mathbf{W}, \mathbf{d}) có thể được tách thành N bài toán nhỏ:

$$\mathcal{L}_1(\mathbf{w}_n, d_n) = \frac{1}{2s} \sum_{m:r_{mn}=1} (\mathbf{x}_m^T \mathbf{w}_n + b_m + d_n - y_{mn})^2 + \frac{\lambda}{2} \|\mathbf{w}_n\|_2^2 \quad (19.5)$$

Mỗi bài toán có thể được tối ưu bằng gradient descent. Công việc quan trọng của chúng ta là tính các đạo hàm của từng hàm mất mát nhỏ này theo \mathbf{w}_n và d_n . Vì biểu thức trong dấu \sum chỉ phụ thuộc vào các *item* đã được đánh giá bởi *user* đang xét (tương ứng với các $r_{mn} = 1$), ta có thể đơn giản (19.5) bằng cách đặt $\hat{\mathbf{X}}_n$ là ma trận con được tạo bởi các cột của \mathbf{X} tương ứng với các *item* đã được đánh giá bởi *user* n , $\hat{\mathbf{b}}_n$ là vector bias con tương ứng, và $\hat{\mathbf{y}}_n$ là các *rating* tương ứng. Khi đó,

$$\mathcal{L}_1(\mathbf{w}_n, d_n) = \frac{1}{2s} \|\hat{\mathbf{X}}_n^T \mathbf{w}_n + \hat{\mathbf{b}}_n + d_n \mathbf{1} - \hat{\mathbf{y}}_n\|^2 + \frac{\lambda}{2} \|\mathbf{w}_n\|_2^2 \quad (19.6)$$

với $\mathbf{1}$ là vector với mọi phần tử bằng 1 và kích thước phù hợp. Đạo hàm của nó là

$$\nabla_{\mathbf{w}_n} \mathcal{L}_1 = \frac{1}{s} \hat{\mathbf{X}}_n (\hat{\mathbf{X}}_n^T \mathbf{w}_n + \hat{\mathbf{b}}_n + d_n \mathbf{1} - \hat{\mathbf{y}}_n) + \lambda \mathbf{w}_n \quad (19.7)$$

$$\nabla_{d_n} \mathcal{L}_1 = \frac{1}{s} \mathbf{1}^T (\hat{\mathbf{X}}_n^T \mathbf{w}_n + \hat{\mathbf{b}}_n + d_n \mathbf{1} - \hat{\mathbf{y}}_n) \quad (19.8)$$

Công thức cập nhật cho \mathbf{w}_n và d_n

$$\mathbf{w}_n \leftarrow \mathbf{w}_n - \eta \left(\frac{1}{s} \hat{\mathbf{X}}_n (\hat{\mathbf{X}}_n^T \mathbf{w}_n + \hat{\mathbf{b}}_n + d_n \mathbf{1} - \hat{\mathbf{y}}_n) + \lambda \mathbf{w}_n \right) \quad (19.9)$$

$$d_n \leftarrow d_n - \eta \left(\frac{1}{s} \mathbf{1}^T (\hat{\mathbf{X}}_n^T \mathbf{w}_n + \hat{\mathbf{b}}_n + d_n \mathbf{1} - \hat{\mathbf{y}}_n) \right) \quad (19.10)$$

Tương tự như thế, mỗi cột \mathbf{x}_m của \mathbf{X} , tức vector đặc trưng cho mỗi *item*, và b_m sẽ được tìm bằng cách tối ưu bài toán

$$\mathcal{L}_2(\mathbf{x}_m, b_m) = \frac{1}{2s} \sum_{n:r_{mn}=1} (\mathbf{w}_n^T \mathbf{x}_m + d_n + b_m - y_{mn})^2 + \frac{\lambda}{2} \|\mathbf{x}_m\|_2^2 \quad (19.11)$$

¹ việc chuẩn hoá sẽ được tự động thực hiện thông qua việc huấn luyện \mathbf{b} và \mathbf{d}

² Bạn đọc có thể thử cộng thêm $\|\mathbf{b}\|_2^2 + \|\mathbf{d}\|_2^2$ vào trong dấu ngoặc của regularization loss. Kết quả có thể thay đổi, nhưng không đáng kể.

Đặt $\hat{\mathbf{W}}_m$ là ma trận được tạo bằng các cột của \mathbf{W} ứng với các *user* đã đánh giá *item* m , $\hat{\mathbf{d}}_m$ là vector con bias tương ứng, và $\hat{\mathbf{y}}^m$ là vector *rating* tương ứng. Bài toán (19.11) trở thành

$$\mathcal{L}(\mathbf{x}_m, b_m) = \frac{1}{2s} \|\hat{\mathbf{W}}_m^T \mathbf{x}_m + \hat{\mathbf{d}}_m + b_n \mathbf{1} - \hat{\mathbf{y}}_m\|^2 + \frac{\lambda}{2} \|\mathbf{x}_m\|_2^2 \quad (19.12)$$

Tương tự như trên, ta có

Công thức cập nhật cho \mathbf{x}_m và b_m

$$\mathbf{x}_m \leftarrow \mathbf{x}_m - \eta \left(\frac{1}{s} \hat{\mathbf{W}}_m (\hat{\mathbf{W}}_m^T \mathbf{x}_m + \hat{\mathbf{d}}_m + b_n \mathbf{1} - \hat{\mathbf{y}}_m) + \lambda \mathbf{x}_m \right) \quad (19.13)$$

$$b_m \leftarrow b_m - \eta \left(\frac{1}{s} \mathbf{1}^T (\hat{\mathbf{W}}_m^T \mathbf{x}_m + \hat{\mathbf{d}}_m + b_n \mathbf{1} - \hat{\mathbf{y}}_m) \right) \quad (19.14)$$

Trong mục tiếp theo, chúng ta sẽ giải quyết bài toán này trên Python.

19.3 Lập trình Python

Trước hết, chúng ta sẽ viết một **class MF** thực hiện việc tối ưu các biến với một ma trận utility được cho dưới dạng **Y_data** giống như với NBCF.

Trước tiên, ta khai báo một vài thư viện cần thiết và khởi tạo **class MF**

```
from __future__ import print_function
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from scipy import sparse

class MF(object):
    def __init__(self, Y, K, lam = 0.1, Xinit = None, Winit = None,
                 learning_rate = 0.5, max_iter = 1000, print_every = 100):
        self.Y = Y # represents the utility matrix
        self.K = K #
        self.lam = lam # regularization parameter
        self.learning_rate = learning_rate # for gradient descent
        self.max_iter = max_iter # maximum number of iterations
        self.print_every = print_every # print loss after each a few iters
        self.n_users = int(np.max(Y[:, 0])) + 1
        self.n_items = int(np.max(Y[:, 1])) + 1
        self.n_ratings = Y.shape[0] # number of known ratings
        self.X = np.random.randn(self.n_items, K) if Xinit is None else Xinit
        self.W = np.random.randn(K, self.n_users) if Winit is None else Winit
        self.b = np.random.randn(self.n_items) # item biases
        self.d = np.random.randn(self.n_users) # user biases
```

Tiếp theo, chúng ta viết các phương thức **loss**, **updateXb**, **updateWd** cho **class MF**.


```

def loss(self):
    L = 0
    for i in range(self.n_ratings):
        # user_id, item_id, rating
        n, m, rating = int(self.Y[i, 0]), int(self.Y[i, 1]), self.Y[i, 2]
        L += 0.5*(self.X[m].dot(self.W[:, n]) + self.b[m] + self.d[n] - rating)**2

    L /= self.n_ratings
    # regularization, don't ever forget this
    return L + 0.5*self.lam*(np.sum(self.X**2) + np.sum(self.W**2))

def updateXb(self):
    for m in range(self.n_items):
        # get all users who rated item m and get the corresponding ratings
        ids = np.where(self.Y[:, 1] == m)[0] # row indices of items m
        user_ids, ratings = self.Y[ids, 0].astype(np.int32), self.Y[ids, 2]
        Wm, dm = self.W[:, user_ids], self.d[user_ids]
        for i in range(30): # 30 iteration for each sub problem
            xm = self.X[m]
            error = xm.dot(Wm) + self.b[m] + dm - ratings
            grad_xm = error.dot(Wm.T)/self.n_ratings + self.lam*xm
            grad_bm = np.sum(error)/self.n_ratings
            # gradient descent
            self.X[m] -= self.learning_rate*grad_xm.reshape(-1)
            self.b[m] -= self.learning_rate*grad_bm

def updateWd(self): # and d
    for n in range(self.n_users):
        # get all items rated by user n, and the corresponding ratings
        ids = np.where(self.Y[:,0] == n)[0] # row indices of items rated by user n
        item_ids, ratings = self.Y[ids, 1].astype(np.int32), self.Y[ids, 2]
        Xn, bn = self.X[item_ids], self.b[item_ids]
        for i in range(30): # 30 iteration for each sub problem
            wn = self.W[:, n]
            error = Xn.dot(wn) + bn + self.d[n] - ratings
            grad_wn = Xn.T.dot(error)/self.n_ratings + self.lam*wn
            grad_dn = np.sum(error)/self.n_ratings
            # gradient descent
            self.W[:, n] -= self.learning_rate*grad_wn.reshape(-1)
            self.d[n] -= self.learning_rate*grad_dn

```

Phần tiếp theo là quá trình tối ưu chính của MF (**fit**), dự đoán *rating* mới (**pred**) và đánh giá chất lượng mô hình bằng root-mean-square error (**evaluate_RMSE**).

```

def fit(self):
    for it in range(self.max_iter):
        self.updateWd()
        self.updateXb()
        if (it + 1) % self.print_every == 0:
            rmse_train = self.evaluate_RMSE(self.Y)
            print('iter = %d, loss = %.4f, RMSE train = %.4f'%(it + 1,
                self.loss(), rmse_train))

```

```

def pred(self, u, i):
    """
    predict the rating of user u for item i
    """
    u, i = int(u), int(i)
    pred = self.X[i, :].dot(self.W[:, u]) + self.b[i] + self.d[u]# + bias
    return max(0, min(5, pred)) # pred should be between 0 and 5 in MoviesLen

def evaluate_RMSE(self, rate_test):
    n_tests = rate_test.shape[0] # number of test
    SE = 0 # squared error
    for n in range(n_tests):
        pred = self.pred(rate_test[n, 0], rate_test[n, 1])
        SE += (pred - rate_test[n, 2])**2

    RMSE = np.sqrt(SE/n_tests)
    return RMSE

```

Tới đây, chúng ta đã xây dựng trong class **MF** với các phương thức cần thiết. Tiếp theo, chúng ta kiểm tra chất lượng mô hình khi nó được áp dụng lên tập dữ liệu MoviesLen 100k.

```

r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings_base = pd.read_csv('ml-100k/ua.base', sep='\t', names=r_cols)
ratings_test = pd.read_csv('ml-100k/ua.test', sep='\t', names=r_cols)

rate_train = ratings_base.as_matrix()
rate_test = ratings_test.as_matrix()

# indices start from 0
rate_train[:, :2] -= 1
rate_test[:, :2] -= 1

rs = MF(rate_train, K = 50, lam = .01, print_every = 5, learning_rate = 50,
        max_iter = 30)
rs.fit()
# evaluate on test data
RMSE = rs.evaluate_RMSE(rate_test)
print('\nMatrix Factorization CF, RMSE = %.4f' %RMSE)

```

Kết quả:

```

iter = 5, loss = 0.4447, RMSE train = 0.9429
iter = 10, loss = 0.4215, RMSE train = 0.9180
iter = 15, loss = 0.4174, RMSE train = 0.9135
iter = 20, loss = 0.4161, RMSE train = 0.9120
iter = 25, loss = 0.4155, RMSE train = 0.9114
iter = 30, loss = 0.4152, RMSE train = 0.9110

Matrix Factorization CF, RMSE = 0.9621

```

RMSE thu được là 0.9621, tốt hơn so với NBCF trong chương trước (0.9688).

19.4 Thảo luận

- **Nonnegative matrix factorization.** Khi dữ liệu chưa được chuẩn hoá, chúng đều mang các giá trị không âm. Kể cả trong trường hợp dải giá trị của *rating* có chứa giá trị âm, ta chỉ cần cộng thêm vào ma trận utility một giá trị hợp lý để có được các *rating* là các số không âm. Khi đó, một phương pháp matrix factorization khác với thêm ràng buộc cũng được sử dụng rất nhiều và mang lại hiệu quả cao trong recommendation system là *nonnegative matrix factorization* (NMF) [ZWFM06], tức phân tích ma trận thành tích các ma trận có các phần tử không âm.

Thông qua matrix factorization, các *user* và *item* được liên kết với nhau bởi các *đặc trưng ẩn*. Độ liên kết của mỗi *user* và *item* tới mỗi đặc trưng ẩn được đo bằng thành phần tương ứng trong vector đặc trưng của chúng, giá trị càng lớn thể hiện việc *user* hoặc *item* có liên quan đến đặc trưng ẩn đó càng lớn. Bằng trực giác, sự liên quan của một *user* hoặc *item* đến một đặc trưng ẩn nên là một số không âm với giá trị 0 thể hiện việc *không liên quan*. Hơn nữa, mỗi *user* và *item* chỉ *liên quan* đến một vài đặc trưng ẩn nhất định. Vì vậy, các vector đặc trưng cho *user* và *item* nên là các vector không âm và có rất nhiều giá trị bằng 0. Những nghiệm này có thể đạt được bằng cách cho thêm ràng buộc không âm vào các thành phần của \mathbf{X} và \mathbf{W} . Đây chính là nguồn gốc của ý tưởng và tên gọi nonnegative matrix factorization.

- **Incremental matrix factorization.** Như đã đề cập, thời gian dự đoán của một recommendation system sử dụng matrix factorization là rất nhanh nhưng thời gian huấn luyện là khá lâu với các tập dữ liệu lớn. Thực tế cho thấy, ma trận utility thay đổi liên tục vì có thêm *user*, *item* cũng như các *rating* mới hoặc *user* muốn thay đổi *rating* của họ, vì vậy các tham số mô hình cũng phải thường xuyên được cập nhật. Điều này đồng nghĩa với việc ta phải tiếp tục thực hiện quá trình *training* vốn tốn khá nhiều thời gian. Việc này được giải quyết phần nào bằng *incremental matrix factorization* [VJG14]. Từ *incremental* có thể được hiểu là *điều chỉnh nhỏ* cho phù hợp với dữ liệu.
- Bài toán tối ưu của matrix factorization có nhiều hướng giải quyết khác ngoài cách áp dụng gradient descent. Bạn đọc có thể xem thêm *Alternating Least Square (ALS)* (<https://goo.gl/g2M4fb>), *Generalized Low Rank Models* (<https://goo.gl/DrDWyW>), và *Singular Value Decomposition* [SKKR02, Pat07]. Chương 20 sẽ bàn kỹ hơn về Singular Value Decomposition.
- Source code trong chương này có thể được tìm thấy tại <https://goo.gl/XbbFH4>.

Dimensionality reduction—Giảm chiều dữ liệu

Số lượng điểm dữ liệu và kích thước của các vector đặc trưng thường rất lớn trong các bài toán thực tế. Nếu thực hiện lưu trữ và tính toán trực tiếp trên dữ liệu có số chiều cao này thì sẽ gặp khó khăn cả về việc lưu trữ và tốc độ tính toán. Vì vậy, *giảm chiều dữ liệu* (*dimensionality reduction* hoặc *dimension reduction*) là một bước quan trọng trong nhiều bài toán machine learning.

Một cách toán học, giảm chiều dữ liệu là việc đi tìm một hàm số $f : \mathbb{R}^D \rightarrow \mathbb{R}^K$ với $K < D$ biến một điểm dữ liệu \mathbf{x} trong không gian có số chiều lớn \mathbb{R}^D thành một điểm \mathbf{z} trong không gian có số chiều nhỏ hơn \mathbb{R}^K . Việc giảm chiều dữ liệu có thể được thực hiện nhằm vào các mục đích khác nhau. Nó có thể phục vụ việc *nép* thông tin sao cho \mathbf{x} có thể được suy ngược lại (xấp xỉ) từ \mathbf{z} . Nó cũng có thể phục vụ các bài toán phân lớp bằng cách chọn ra những đặc trưng quan trọng (*feature selection*) hoặc tạo ra các đặc trưng mới từ đặc trưng cũ (*feature extraction*) sao cho kết quả của bài toán phân lớp được cải thiện.

Trong nhiều trường hợp, làm việc trên dữ liệu được giảm chiều cho kết quả tốt hơn dữ liệu trong không gian ban đầu.

Trong phần này, chúng ta sẽ xem xét các phương pháp giảm chiều dữ liệu phổ biến nhất: *principle component analysis* cho bài toán giảm chiều dữ liệu vẫn giữ được tối đa lượng thông tin, và *linear discriminant analysis* cho bài toán giữ lại những đặc trưng quan trọng nhất cho việc phân lớp. Trước hết, chúng ta cùng tìm hiểu một phương pháp phân tích ma trận thành nhân tử vô cùng quan trọng – *singular value decomposition*.

Singular value decomposition

20.1 Giới thiệu

Nhắc lại bài toán chéo hoá ma trận: Một ma trận vuông $\mathbf{A} \in \mathbb{R}^{n \times n}$ được gọi là *chéo hoá được* (*diagonalizable*) nếu tồn tại ma trận đường chéo \mathbf{D} và ma trận khả nghịch \mathbf{P} sao cho:

$$\mathbf{A} = \mathbf{P}\mathbf{D}\mathbf{P}^{-1} \quad (20.1)$$

Số lượng phần tử khác 0 của ma trận đường chéo \mathbf{D} chính là rank của ma trận \mathbf{A} .

Nhân cả hai vế của (20.1) với \mathbf{P} ta có:

$$\mathbf{A}\mathbf{P} = \mathbf{P}\mathbf{D} \quad (20.2)$$

Gọi $\mathbf{p}_i, \mathbf{d}_i$ lần lượt là cột thứ i của ma trận \mathbf{P} và \mathbf{D} . Vì mỗi một cột của vế trái và vế phải của (20.2) phải bằng nhau, ta cần có

$$\mathbf{A}\mathbf{p}_i = \mathbf{P}\mathbf{d}_i = d_{ii}\mathbf{p}_i \quad (20.3)$$

với d_{ii} là phần tử thứ i của \mathbf{d}_i . Dấu bằng thứ hai xảy ra vì \mathbf{D} là ma trận đường chéo, tức \mathbf{d}_i chỉ có thành phần d_{ii} là khác 0. Biểu thức (20.3) chỉ ra rằng mỗi phần tử d_{ii} phải là một trị riêng của \mathbf{A} và mỗi vector cột \mathbf{p}_i phải là một vector riêng của \mathbf{A} ứng với trị riêng d_{ii} .

Cách phân tích một ma trận vuông thành nhân tử như (20.1) còn được gọi là *Eigen Decomposition*. Một điểm quan trọng là cách phân tích này chỉ được áp dụng với ma trận vuông và không phải lúc nào cũng tồn tại. Nó chỉ tồn tại nếu ma trận \mathbf{A} có n vector riêng độc lập tuyến tính, vì nếu không thì không tồn tại ma trận \mathbf{P} khả nghịch. Thêm nữa, cách phân tích này cũng không phải là duy nhất vì nếu \mathbf{P}, \mathbf{D} thoả mãn (20.1) thì $k\mathbf{P}, \mathbf{D}$ cũng thoả mãn với k là một số thực khác 0 bất kỳ.

Việc phân tích một ma trận ra thành tích của nhiều ma trận đặc biệt khác (*matrix factorization* hoặc *matrix decomposition*) mang lại nhiều ích lợi quan trọng mà các bạn sẽ thấy:

giảm số chiều dữ liệu, nén dữ liệu, tìm hiểu các đặc tính của dữ liệu, giải các hệ phương trình tuyến tính, clustering, và nhiều ứng dụng khác. Hệ thống khuyến nghị cũng là một trong rất nhiều ứng dụng của matrix factorization.

Trong chương này, chúng ta sẽ làm quen với một trong những phương pháp matrix factorization rất đẹp của đại số tuyến tính có tên là *singular value decomposition* (SVD) [GR70]. Các bạn sẽ thấy, mọi ma trận, không nhất thiết là vuông, đều có thể được phân tích thành tích của ba ma trận đặc biệt.

20.2 Singular value decomposition

Để hạn chế nhầm lẫn trong các phép toán nhân ma trận, chúng ta cần để ý tới kích thước của mỗi ma trận. Trong chương này, ta sẽ ký hiệu một ma trận cùng với số chiều của nó, ví dụ $\mathbf{A}_{m \times n}$ dùng để ký hiệu một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$.

20.2.1 Phát biểu SVD

Singular value decomposition

Một ma trận $\mathbf{A}_{m \times n}$ bất kỳ đều có thể phân tích thành dạng:

$$\mathbf{A}_{m \times n} = \mathbf{U}_{m \times m} \mathbf{\Sigma}_{m \times n} (\mathbf{V}_{n \times n})^T \quad (20.4)$$

Trong đó, \mathbf{U}, \mathbf{V} là các ma trận trực giao, $\mathbf{\Sigma}$ là một ma trận đường chéo cùng kích thước với \mathbf{A} . Các phần tử trên đường chéo chính của $\mathbf{\Sigma}$ là không âm và được theo thứ tự giảm dần $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \geq 0 = 0 = \dots = 0$. Số lượng các phần tử khác 0 trong $\mathbf{\Sigma}$ chính là rank của ma trận \mathbf{A} : $r = \text{rank}(\mathbf{A})$.

SVD của một ma trận bất kỳ luôn tồn tại. Bạn đọc có thể tìm thấy chứng minh cho việc này tại <https://goo.gl/TdtWDQ>. Cách biểu diễn (20.4) không là duy nhất vì ta chỉ cần đổi dấu của cả \mathbf{U} và \mathbf{V} thì (20.4) vẫn thỏa mãn.

Hình 20.1 mô tả SVD của ma trận $\mathbf{A}_{m \times n}$ trong hai trường hợp: $m < n$ và $m > n$. Trường hợp $m = n$ có thể xếp vào một trong hai trường hợp trên.

20.2.2 Nguồn gốc tên gọi singular value decomposition

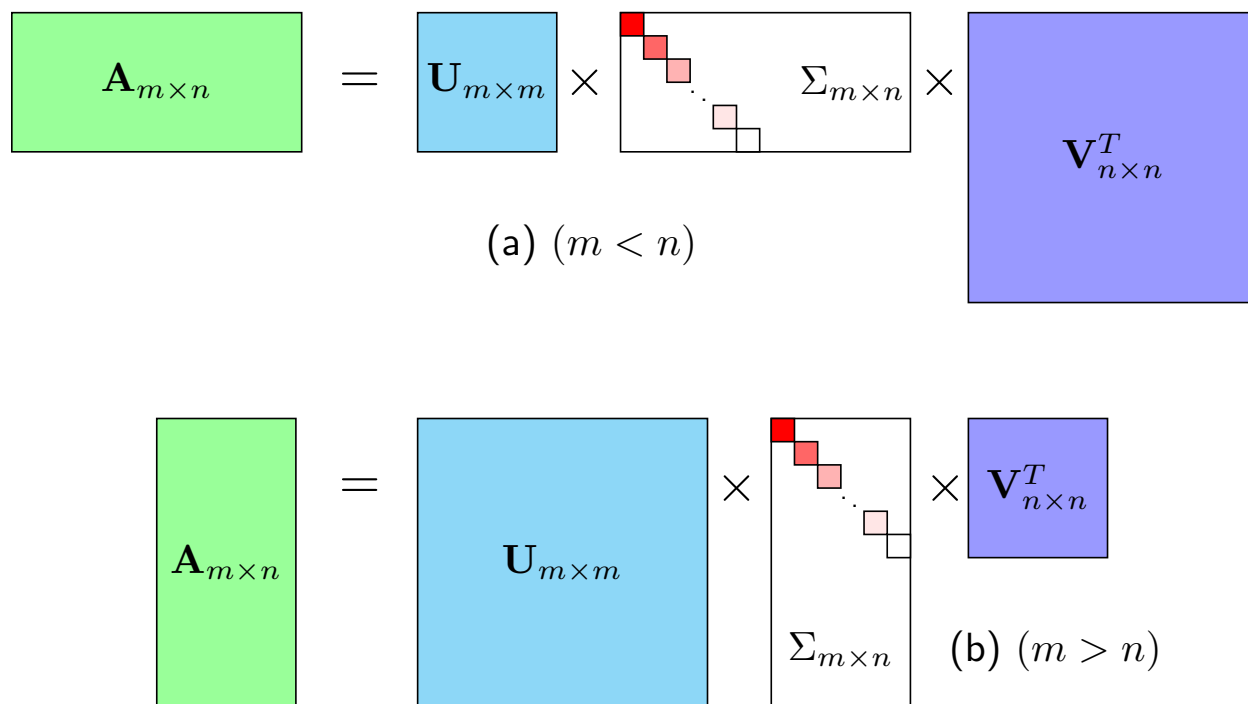
Tạm bỏ qua chiều của mỗi ma trận, từ (20.4) ta có:

$$\mathbf{A}\mathbf{A}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T)^T \quad (20.5)$$

$$= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V}\mathbf{\Sigma}^T\mathbf{U}^T \quad (20.6)$$

$$= \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^T = \mathbf{U}\mathbf{\Sigma}\mathbf{\Sigma}^T\mathbf{U}^{-1} \quad (20.7)$$

Dấu bằng ở (20.6) xảy ra vì $\mathbf{V}^T\mathbf{V} = \mathbf{I}$ do \mathbf{V} là một ma trận trực giao. Dấu bằng ở (20.7) xảy ra vì \mathbf{U} là một ma trận trực giao.



Hình 20.1: SVD cho ma trận \mathbf{A} khi: $m < n$ (hình trên), và $m > n$ (hình dưới). Σ là một ma trận đường chéo với các phần tử trên đó giảm dần và không âm. Màu đỏ càng đậm thể hiện giá trị càng cao. Các ô màu trắng trên ma trận này thể hiện giá trị 0.

Quan sát thấy rằng $\Sigma \Sigma^T$ là một ma trận đường chéo với các phần tử trên đường chéo là $\sigma_1^2, \sigma_2^2, \dots$. Vậy (20.7) chính là một eigen decomposition của $\mathbf{A} \mathbf{A}^T$. Thêm nữa, $\sigma_1^2, \sigma_2^2, \dots$ chính là các trị riêng của $\mathbf{A} \mathbf{A}^T$. Ma trận $\mathbf{A} \mathbf{A}^T$ luôn là ma trận nửa xác định dương nên các trị riêng của nó là không âm. Các σ_i , là căn bậc hai của các trị riêng của $\mathbf{A} \mathbf{A}^T$, còn được gọi là *singular value* của \mathbf{A} . Tên gọi *singular value decomposition* xuất phát từ đây.

Cũng theo đó, mỗi cột của \mathbf{U} chính là một vector riêng của $\mathbf{A} \mathbf{A}^T$. Ta gọi mỗi cột này là một *left-singular vector* của \mathbf{A} . Tương tự như thế, $\mathbf{A}^T \mathbf{A} = \mathbf{V} \Sigma^T \Sigma \mathbf{V}^T$ và các cột của \mathbf{V} còn được gọi là các *right-singular vectors* của \mathbf{A} .

Trong Python, để tính SVD của một ma trận, chúng ta sử dụng module `linalg` của `numpy`:

```

from __future__ import print_function
import numpy as np
from numpy import linalg as LA

m, n = 3, 4
A = np.random.rand(m, n)
U, S, V = LA.svd(A) # A = U*S*V (no V transpose here)
# checking if U, V are orthogonal and S is a diagonal matrix with
# nonnegative decreasing elements
print('Frobenius norm of (UU^T - I) =', LA.norm(U.dot(U.T) - np.eye(m)))
print('S = ', S)
print('Frobenius norm of (VV^T - I) =', LA.norm(V.dot(V.T) - np.eye(n)))
    
```


Kết quả:

```
Frobenius norm of (UU^T - I) = 4.09460889695e-16
S = [ 1.76321041  0.59018069  0.3878011 ]
Frobenius norm of (VV^T - I) = 5.00370755311e-16
```

Lưu ý rằng biến \mathbf{s} được trả về chỉ bao gồm các phần tử trên đường chéo của $\mathbf{\Sigma}$. Biến \mathbf{v} trả về là \mathbf{V}^T trong (20.4).

20.2.3 Singular value của một ma trận nửa xác định dương

Giả sử \mathbf{A} là một ma trận đối xứng vuông nửa xác định dương, ta sẽ chứng minh rằng các singular value của \mathbf{A} chính là các trị riêng của nó. Thật vậy, gọi λ là một trị riêng của \mathbf{A} và \mathbf{x} là một vector riêng ứng với trị riêng đó, hơn nữa $\|\mathbf{x}\|_2 = 1$. Vì \mathbf{A} là nửa xác định dương, ta phải có $\lambda \geq 0$. Ta có

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \Rightarrow \mathbf{A}^T\mathbf{A}\mathbf{x} = \lambda\mathbf{A}\mathbf{x} = \lambda^2\mathbf{x} \quad (20.8)$$

Như vậy, λ^2 là một trị riêng của $\mathbf{A}^T\mathbf{A} \Rightarrow$ singular value của \mathbf{A} chính là $\sqrt{\lambda^2} = \lambda$.

20.2.4 Compact SVD

Viết lại biểu thức (20.4) dưới dạng tổng của các ma trận có rank bằng 1:

$$\mathbf{A} = \sigma_1\mathbf{u}_1\mathbf{v}_1^T + \sigma_2\mathbf{u}_2\mathbf{v}_2^T + \dots + \sigma_r\mathbf{u}_r\mathbf{v}_r^T \quad (20.9)$$

với chú ý rằng mỗi $\mathbf{u}_i\mathbf{v}_i^T, 1 \leq i \leq r$, là một ma trận có rank bằng 1.

Rõ ràng trong cách biểu diễn này, ma trận \mathbf{A} chỉ phụ thuộc vào r cột đầu tiên của \mathbf{U}, \mathbf{V} và r giá trị khác 0 trên đường chéo của ma trận $\mathbf{\Sigma}$. Vì vậy ta có một cách phân tích gọn hơn và gọi là *compact SVD*:

$$\mathbf{A} = \mathbf{U}_r\mathbf{\Sigma}_r(\mathbf{V}_r)^T \quad (20.10)$$

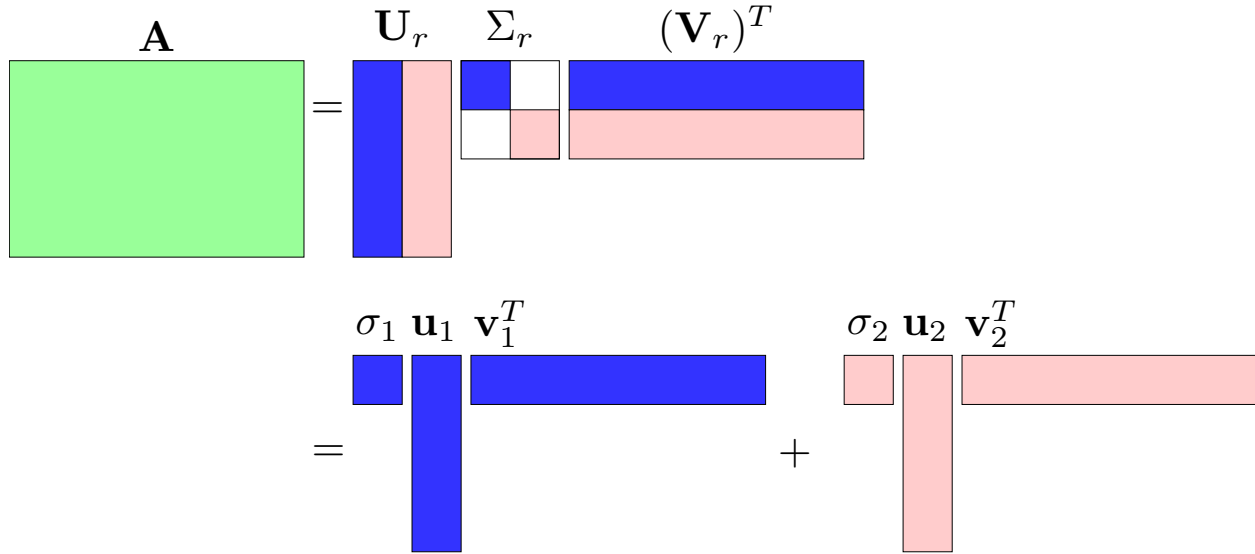
với $\mathbf{U}_r, \mathbf{V}_r$ lần lượt là ma trận được tạo bởi r cột đầu tiên của \mathbf{U} và \mathbf{V} . $\mathbf{\Sigma}_r$ là ma trận con được tạo bởi r hàng đầu tiên và r cột đầu tiên của $\mathbf{\Sigma}$. Nếu ma trận \mathbf{A} có rank nhỏ hơn rất nhiều so với số hàng và số cột $r \ll m, n$, ta sẽ được lợi nhiều về việc lưu trữ.

Dưới đây là ví dụ minh họa với $m = 4, n = 6, r = 2$.

20.2.5 Truncated SVD

Nhắc lại rằng trong ma trận $\mathbf{\Sigma}$, các giá trị trên đường chéo là không âm và giảm dần $\sigma_1 \geq \sigma_2 \geq \dots, \geq \sigma_r \geq 0 = 0 = \dots = 0$. Thông thường, chỉ một lượng nhỏ các σ_i mang giá trị lớn, các giá trị còn lại thường nhỏ và gần 0. Khi đó ta có thể xấp xỉ ma trận \mathbf{A} bằng tổng của $k < r$ ma trận có rank 1:

$$\mathbf{A} \approx \mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k(\mathbf{V}_k)^T = \sigma_1\mathbf{u}_1\mathbf{v}_1^T + \sigma_2\mathbf{u}_2\mathbf{v}_2^T + \dots + \sigma_k\mathbf{u}_k\mathbf{v}_k^T \quad (20.11)$$



Hình 20.2: Biểu diễn SVD dạng thu gọn và biểu diễn ma trận dưới dạng tổng các ma trận có rank bằng 1. Các khối ma trận đặt cạnh nhau thể hiện phép nhân ma trận.

Dưới đây là một định lý thú vị. Định lý này nói rằng sai số do cách xấp xỉ trên chính là căn bậc hai của tổng bình phương của các singular value mà ta đã bỏ qua ở phần cuối của Σ . Ở đây sai số được định nghĩa là Frobenius norm của hiệu hai ma trận.

Định lý 20.1: Sai số do xấp xỉ bởi truncated SVD

Nếu xấp xỉ một ma trận \mathbf{A} có rank r bởi truncated SVD với $k < r$ phần tử, sai số do cách xấp xỉ này là

$$\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i=k+1}^r \sigma_i^2 \quad (8) \quad (20.12)$$

Chứng minh: Sử dụng tính chất $\|\mathbf{X}\|_F^2 = \text{trace}(\mathbf{X}\mathbf{X}^T)$ và $\text{trace}(\mathbf{X}\mathbf{Y}) = \text{trace}(\mathbf{Y}\mathbf{X})$ với mọi ma trận \mathbf{X}, \mathbf{Y} ta có

$$\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \left\| \sum_{i=k+1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^T \right\|_F^2 = \text{trace} \left\{ \left(\sum_{i=k+1}^r \sigma_i \mathbf{u}_i \mathbf{v}_i^T \right) \left(\sum_{j=k+1}^r \sigma_j \mathbf{u}_j \mathbf{v}_j^T \right)^T \right\} \quad (20.13)$$

$$= \text{trace} \left\{ \sum_{i=k+1}^r \sum_{j=k+1}^r \sigma_i \sigma_j \mathbf{u}_i \mathbf{v}_i^T \mathbf{v}_j \mathbf{u}_j^T \right\} = \text{trace} \left\{ \sum_{i=k+1}^r \sigma_i^2 \mathbf{u}_i \mathbf{u}_i^T \right\} \quad (20.14)$$

$$= \text{trace} \left\{ \sum_{i=k+1}^r \sigma_i^2 \mathbf{u}_i^T \mathbf{u}_i \right\} \quad (20.15)$$

$$= \text{trace} \left\{ \sum_{i=k+1}^r \sigma_i^2 \right\} = \sum_{i=k+1}^r \sigma_i^2 \quad (20.16)$$

Dấu bằng thứ hai ở (20.14) xảy ra vì \mathbf{V} có các cột vuông góc với nhau. Dấu bằng ở (20.15) xảy ra vì hàm trace có tính chất giao hoán. Dấu bằng ở (20.16) xảy ra vì biểu thức trong dấu ngoặc là một số vô hướng. \square

Thay $k = 0$ ta sẽ có

$$\|\mathbf{A}\|_F^2 = \sum_{i=1}^r \sigma_i^2 \quad (20.17)$$

Từ đó

$$\frac{\|\mathbf{A} - \mathbf{A}_k\|_F^2}{\|\mathbf{A}\|_F^2} = \frac{\sum_{i=k+1}^r \sigma_i^2}{\sum_{j=1}^r \sigma_j^2} \quad (20.18)$$

Như vậy, sai số do xấp xỉ càng nhỏ nếu các singular value bị *truncated* có giá trị càng nhỏ so với các singular value được giữ lại. Đây là một định lý quan trọng giúp xác định việc xấp xỉ ma trận dựa trên lượng thông tin muốn giữ lại. Với giả sử rằng *lượng thông tin* được định nghĩa là tổng bình phương của các singular value. Ví dụ, nếu ta muốn giữ lại ít nhất 90% lượng thông tin trong \mathbf{A} , trước hết ta tính $\sum_{j=1}^r \sigma_j^2$, sau đó chọn k là số nhỏ nhất sao cho

$$\frac{\sum_{i=1}^k \sigma_i^2}{\sum_{j=1}^r \sigma_j^2} \geq 0.9 \quad (20.19)$$

Khi k nhỏ, ma trận \mathbf{A}_k có rank là k , là một ma trận có rank nhỏ. Vì vậy, Truncated SVD còn được coi là một phương pháp *low-rank approximation*.

20.2.6 Xấp xỉ rank k tốt nhất

Người ta chứng minh được rằng¹ \mathbf{A}_k chính là nghiệm của bài toán tối ưu sau đây:

$$\begin{aligned} & \min_{\mathbf{B}} \|\mathbf{A} - \mathbf{B}\|_F \\ & \text{thoả mãn: rank}(\mathbf{B}) = k \end{aligned} \quad (20.20)$$

và như đã chứng minh ở trên $\|\mathbf{A} - \mathbf{A}_k\|_F^2 = \sum_{i=k+1}^r \sigma_i^2$.

Nếu sử dụng ℓ_2 norm của ma trận (xem Phụ lục A) thay vì Frobenius norm để đo sai số, \mathbf{A}_k cũng là nghiệm của bài toán tối ưu

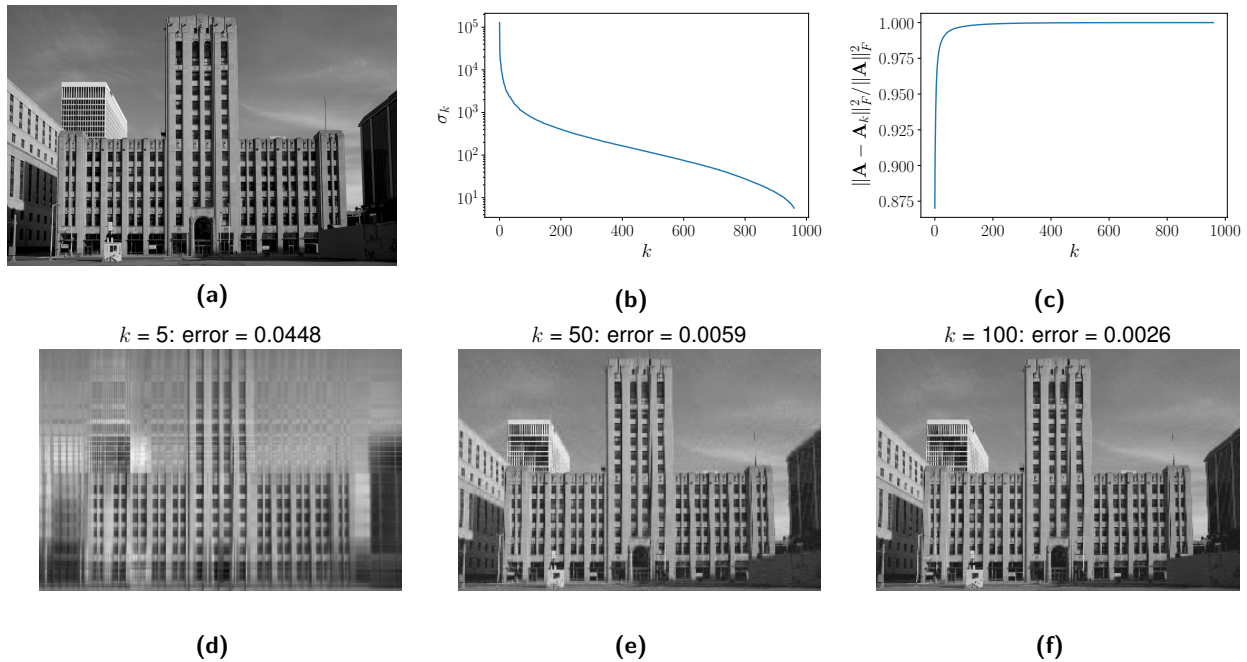
$$\begin{aligned} & \min_{\mathbf{B}} \|\mathbf{A} - \mathbf{B}\|_2 \\ & \text{thoả mãn: rank}(\mathbf{B}) = k \end{aligned} \quad (20.21)$$

và sai số $\|\mathbf{A} - \mathbf{A}_k\|_2^2 = \sigma_{k+1}^2$. Trong đó, norm 2 của một ma trận được định nghĩa bởi

$$\|\mathbf{A}\|_2 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2 \quad (20.22)$$

Frobenius norm và ℓ_2 norm là hai norm được sử dụng nhiều nhất trong ma trận. Như vậy, xét trên cả hai norm này, truncated SVD đều cho xấp xỉ tốt nhất. Vì vậy, truncated SVD còn được coi là *xấp xỉ rank thấp tốt nhất (best low-rank approximation)*.

¹ Singular Value Decomposition – Princeton (<https://goo.gl/hU38GF>).



Hình 20.3: Ví dụ về SVD cho ảnh. (a) Bức ảnh gốc là một ảnh xám, là một ma trận cỡ 960×1440 . (b) Giá trị của các singular values của ma trận ảnh theo logscale. Có thể thấy rằng các singular value giảm nhanh ở khoảng $k = 200$. (c) Biểu diễn lượng thông tin được giữ lại khi chọn các k khác nhau. Có thể nhận thấy từ khoảng $k = 200$, lượng thông tin giữ lại là gần bằng 1. Vậy ta có thể xấp xỉ ma trận ảnh này bằng một ma trận có rank nhỏ hơn. (d), (e), (f) Các ảnh xấp xỉ với k lần lượt là 5, 50, 100.

20.3 SVD cho image compression

Xét ví dụ trong Hình 20.3. Bức ảnh gốc trong Hình 20.3a là một ảnh xám có kích thước 960×1440 pixel. Bức ảnh này có thể được coi là một ma trận $\mathbf{A} \in \mathbb{R}^{960 \times 1440}$. Ta có thể quan sát thấy rằng ma trận này là *low-rank* vì rất nhiều *tầng* của toà nhà nhìn tương tự nhau. Hình 20.3b là giá trị của các singular value của bức ảnh được sắp xếp theo thứ tự giảm dần. Chú ý rằng giá trị của các singular value được biểu diễn trên thang log10 nên các giá trị singular đầu tiên rất lớn so với các giá trị singular ở cuối. Hình 20.3c mô tả chất lượng của việc xấp xỉ \mathbf{A} bởi \mathbf{A}_k bằng truncated SVD. Ta cũng thấy rằng giá trị này xấp xỉ bằng 1 tại $k = 200$. Hình 20.3d, 20.3e, 20.3f là các bức ảnh xấp xỉ khi chọn các giá trị k khác nhau. Khi k gần 100, lượng thông tin mất đi rơi vào khoảng nhỏ hơn 3%, ảnh thu được có chất lượng gần như ảnh gốc.

Để lưu ảnh với truncated SVD, ta sẽ lưu các ma trận $\mathbf{U}_k \in \mathbb{R}^{m \times k}$, $\mathbf{\Sigma}_k \in \mathbb{R}^{k \times k}$, $\mathbf{V}_k \in \mathbb{R}^{n \times k}$. Tổng số phần tử phải lưu là $k(m + n + 1)$ với chú ý rằng ta chỉ cần lưu các giá trị trên đường chéo của $\mathbf{\Sigma}_k$. Giả sử mỗi phần tử được lưu bởi một số thực bốn byte, thế thì số byte cần lưu trừ là $4k(m + n + 1)$. Nếu so giá trị này với ảnh gốc có kích thước mn , mỗi giá trị là một số nguyên một byte, tỉ lệ nén là

$$\frac{4k(m + n + 1)}{mn} \tag{20.23}$$

Khi $k \ll m, n$, ta được một tỉ lệ nhỏ hơn 1. Trong ví dụ trên, $m = 960, n = 1440, k = 100$, tỉ lệ nén là xấp xỉ 0.69, tức đã tiết kiệm được khoảng 30% bộ nhớ.

20.4 Thảo luận

- Ngoài ứng dụng nêu trên, SVD còn được ứng dụng trong việc giải phương trình tuyến tính thông qua giả nghịch đảo Moore Penrose (<https://goo.gl/4wrXue>), recommendation system [SKKR00], dimensionality reduction [Cyb89], image deblurring [HNO06], clustering [DFK+04], v.v..
- Khi ma trận \mathbf{A} lớn, việc tính toán SVD của nó tốn nhiều thời gian. Cách tính Truncated SVD với k nhỏ bằng cách tính SVD như được sử dụng trở nên không khả thi. Thay vào đó, có một phương pháp lặp giúp tính các trị riêng và vector riêng của một ma trận lớn một cách hiệu quả, và ta chỉ cần tìm k trị riêng lớn nhất của $\mathbf{A}\mathbf{A}^T$ và các vector riêng tương ứng, việc này sẽ tiết kiệm được khá nhiều thời gian. Bạn đọc có thể tìm đọc thêm *Power method for approximating eigenvalues* (<https://goo.gl/PfDqsn>).
- Source code trong chương này có thể được tìm thấy tại <https://goo.gl/Z3wbsU>.

Đọc thêm

1. *Singular Value Decomposition - Stanford University* (<https://goo.gl/Gp726X>).
2. *Singular Value Decomposition - Princeton* (<https://goo.gl/HKpcsB>).
3. *CS168: The Modern Algorithmic Toolbox Lecture #9: The Singular Value Decomposition (SVD) and Low-Rank Matrix Approximations - Stanford* (<https://goo.gl/RV57KU>).
4. *The Moore-Penrose Pseudoinverse (Math 33A - UCLA)* (<https://goo.gl/VxMYx1>).

Principal component analysis

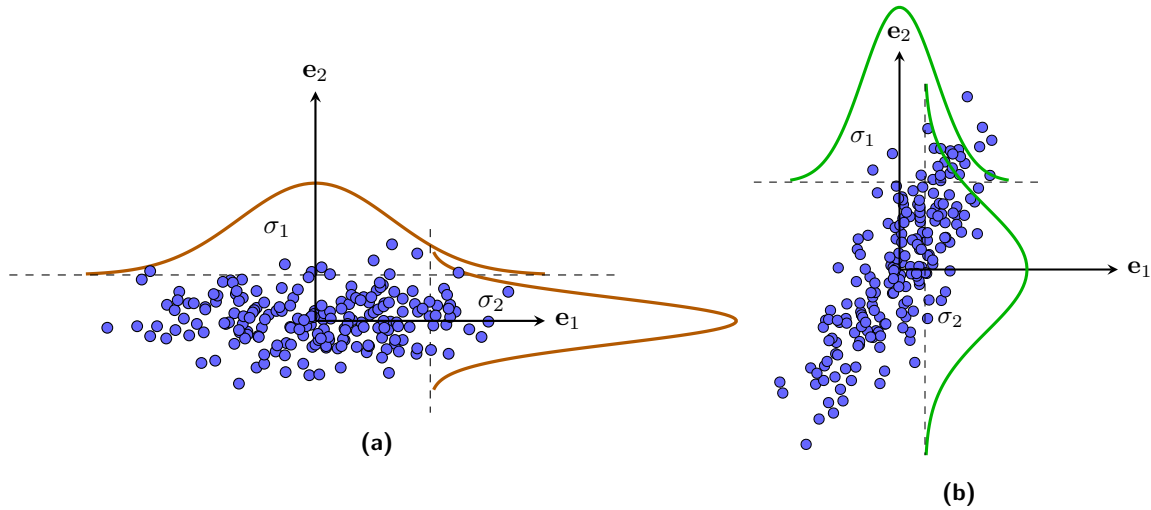
21.1 Principal component analysis

21.1.1 Ý tưởng

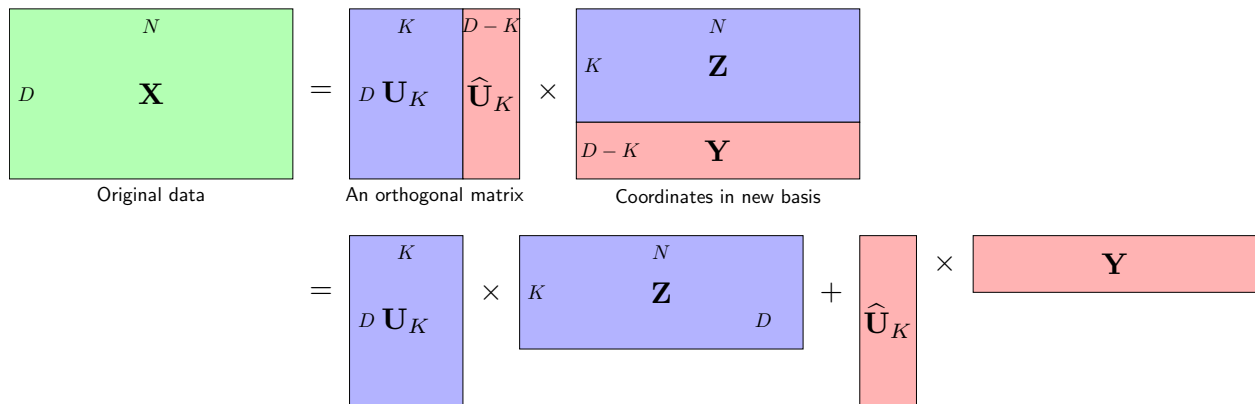
Giả sử dữ liệu ban đầu là $\mathbf{x} \in \mathbb{R}^D$ và dữ liệu đã được giảm chiều là $\mathbf{z} \in \mathbb{R}^K$ với $K < D$. Cách đơn giản nhất để giảm chiều dữ liệu từ D về $K < D$ là chỉ giữ lại K phần tử *quan trọng nhất*. Có hai câu hỏi lập tức được đặt ra ở đây. Thứ nhất, làm thế nào để xác định *tầm quan trọng* của mỗi chiều dữ liệu? Thứ hai, nếu tầm quan trọng của các chiều dữ liệu là như nhau, ta cần bỏ đi những chiều nào?

Để trả lời câu hỏi thứ nhất, ta hãy quan sát Hình 21.1a. Giả sử các điểm dữ liệu có thành phần thứ hai (phương đứng) giống hệt nhau hoặc sai khác nhau rất ít (phương sai nhỏ). Như vậy, thành phần này hoàn toàn có thể được lược bỏ đi, và ta ngầm hiểu rằng nó sẽ được xấp xỉ bằng kỳ vọng của thành phần đó trên toàn bộ các điểm dữ liệu. Ngược lại, việc làm này nếu được áp dụng lên thành phần thứ nhất (phương ngang) sẽ khiến *lượng thông tin* bị mất đi rất nhiều do sai số xấp xỉ là quá lớn. Lượng thông tin theo mỗi thành phần, vì vậy, có thể được đo bằng phương sai của dữ liệu trên thành phần đó. Tổng lượng thông tin có thể được coi là tổng phương sai trên toàn bộ các thành phần. Lấy một ví dụ về việc có hai camera được đặt dùng để chụp một con người, một camera đặt phía trước người và một camera đặt trên đầu. Rõ ràng, hình ảnh thu được từ camera đặt phía trước người mang nhiều thông tin hơn so với hình ảnh nhìn từ phía trên đầu. Vì vậy, bức ảnh chụp từ phía trên đầu có thể được bỏ qua mà không có quá nhiều thông tin về hình dáng của người đó bị mất.

Câu hỏi thứ hai tương ứng với trường hợp Hình 21.1b. Trong cả hai chiều, phương sai của dữ liệu đều lớn; việc bỏ đi một trong hai chiều đều dẫn đến việc lượng thông tin bị mất đi là lớn. Tuy nhiên, quan sát ban đầu của chúng ta là nếu xoay trục tọa độ đi một góc phù hợp, một trong hai chiều dữ liệu có thể được giảm đi vì dữ liệu có xu hướng phân bố xung quanh một đường thẳng.



Hình 21.1: Ví dụ về phương sai của dữ liệu trong không gian hai chiều. (a) Chiều thứ hai có phương sai (tỉ lệ với độ rộng của đường hình chuông) nhỏ hơn chiều thứ nhất. (b) Cả hai chiều có phương sai đáng kể. Phương sai của mỗi chiều là phương sai của thành phần tương ứng được lấy trên toàn bộ dữ liệu. Phương sai tỉ lệ thuận với độ phân tán của dữ liệu.



Hình 21.2: Ý tưởng chính của PCA: Tìm một hệ trục chuẩn mới sao cho trong hệ này, các thành phần quan trọng nhất nằm trong K thành phần đầu tiên.

Principle component analysis (PCA) là một phương pháp đi tìm một phép xoay trục tọa độ để được một hệ trục tọa độ mới sao cho trong hệ mới này, thông tin của dữ liệu chủ yếu tập trung ở một vài thành phần. Phần còn lại chứa ít thông tin hơn có thể được lược bỏ.

Phép xoay trục tọa độ có liên hệ chặt chẽ tới hệ trục chuẩn và ma trận trực giao (xem Mục 1.9 và 1.10). Giả sử hệ cơ sở trục chuẩn mới là \mathbf{U} (mỗi cột của \mathbf{U} là một vector đơn vị cho một chiều) và chúng ta muốn giữ lại K tọa độ trong hệ cơ sở mới này. Không mất tính tổng quát, giả sử đó là K thành phần đầu tiên. Quan sát Hình 21.2 với cơ sở mới $\mathbf{U} = [\mathbf{U}_K, \hat{\mathbf{U}}_K]$ là một hệ trục chuẩn với \mathbf{U}_K là ma trận con tạo bởi K cột đầu tiên của \mathbf{U} . Với cơ sở mới này, ma trận dữ liệu có thể được viết thành

$$\mathbf{X} = \mathbf{U}_K \mathbf{Z} + \hat{\mathbf{U}}_K \mathbf{Y} \quad (21.1)$$

Từ đây ta cũng suy ra

$$\begin{bmatrix} \mathbf{Z} \\ \mathbf{Y} \end{bmatrix} = \begin{bmatrix} \mathbf{U}_K^T \\ \widehat{\mathbf{U}}_K^T \end{bmatrix} \mathbf{X} \Rightarrow \begin{aligned} \mathbf{Z} &= \mathbf{U}_K^T \mathbf{X} \\ \mathbf{Y} &= \widehat{\mathbf{U}}_K^T \mathbf{X} \end{aligned} \quad (21.2)$$

Mục đích của PCA là đi tìm ma trận trực giao \mathbf{U} sao cho phần lớn thông tin được giữ lại ở phần màu xanh $\mathbf{U}_K \mathbf{Z}$ và phần màu đỏ $\widehat{\mathbf{U}}_K \mathbf{Y}$ sẽ được lược bỏ và thay bằng một ma trận không phụ thuộc vào từng điểm dữ liệu. Cụ thể, ta sẽ xấp xỉ \mathbf{Y} bởi một ma trận có toàn bộ các cột là như nhau. *Chú ý rằng các cột này có thể phụ thuộc vào dữ liệu huấn luyện nhưng không phụ thuộc vào dữ liệu kiểm thử.* Gọi mỗi cột đó là \mathbf{b} và có thể coi nó là bias, khi đó, ta sẽ xấp xỉ $\mathbf{Y} \approx \mathbf{b} \mathbf{1}^T$, trong đó $\mathbf{1}^T \in \mathbb{R}^{1 \times N}$ là một vector hàng có toàn bộ các phần tử bằng 1. Giả sử đã tìm được \mathbf{U} , ta cần tìm \mathbf{b} thoả mãn:

$$\mathbf{b} = \operatorname{argmin}_{\mathbf{b}} \|\mathbf{Y} - \mathbf{b} \mathbf{1}^T\|_F^2 = \operatorname{argmin}_{\mathbf{b}} \|\widehat{\mathbf{U}}_K^T \mathbf{X} - \mathbf{b} \mathbf{1}^T\|_F^2 \quad (21.3)$$

Giải phương trình đạo hàm theo \mathbf{b} của hàm mục tiêu bằng $\mathbf{0}$:

$$(\mathbf{b} \mathbf{1}^T - \widehat{\mathbf{U}}_K^T \mathbf{X}) \mathbf{1} = \mathbf{0} \Rightarrow N \mathbf{b} = \widehat{\mathbf{U}}_K^T \mathbf{X} \mathbf{1} \Rightarrow \mathbf{b} = \widehat{\mathbf{U}}_K^T \bar{\mathbf{x}} \quad (21.4)$$

ở đây ta đã sử dụng $\mathbf{1}^T \mathbf{1} = N$ và $\bar{\mathbf{x}} = \frac{1}{N} \mathbf{X} \mathbf{1}$ là vector trung bình của toàn bộ các cột của \mathbf{X} . Với giá trị \mathbf{b} tìm được này, dữ liệu ban đầu sẽ được xấp xỉ bởi

$$\mathbf{X} = \mathbf{U}_K \mathbf{Z} + \widehat{\mathbf{U}}_K \mathbf{Y} \approx \mathbf{U}_K \mathbf{Z} + \widehat{\mathbf{U}}_K \mathbf{b} \mathbf{1}^T = \mathbf{U}_K \mathbf{Z} + \widehat{\mathbf{U}}_K \widehat{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T \triangleq \tilde{\mathbf{X}} \quad (21.5)$$

21.1.2 Hàm mất mát

Hàm mất mát của PCA có thể được coi như sai số của phép xấp xỉ, và được định nghĩa là

$$\begin{aligned} \frac{1}{N} \|\mathbf{X} - \tilde{\mathbf{X}}\|_F^2 &= \frac{1}{N} \|\widehat{\mathbf{U}}_K \mathbf{Y} - \widehat{\mathbf{U}}_K \widehat{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T\|_F^2 = \frac{1}{N} \|\widehat{\mathbf{U}}_K \widehat{\mathbf{U}}_K^T \mathbf{X} - \widehat{\mathbf{U}}_K \widehat{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T\|_F^2 \\ &= \frac{1}{N} \|\widehat{\mathbf{U}}_K \widehat{\mathbf{U}}_K^T (\mathbf{X} - \bar{\mathbf{x}} \mathbf{1}^T)\|_F^2 \triangleq J \end{aligned} \quad (21.6)$$

Chú ý rằng, nếu các cột của một ma trận \mathbf{V} bất kỳ tạo thành một hệ trực chuẩn thì với một ma trận \mathbf{W} bất kỳ, ta luôn có

$$\|\mathbf{V} \mathbf{W}\|_F^2 = \operatorname{trace}(\mathbf{W}^T \mathbf{V}^T \mathbf{V} \mathbf{W}) = \operatorname{trace}(\mathbf{W}^T \mathbf{W}) = \|\mathbf{W}\|_F^2 \quad (21.7)$$

Đặt $\widehat{\mathbf{X}} = \mathbf{X} - \bar{\mathbf{x}} \mathbf{1}^T$. Ma trận này có được bằng cách trừ mỗi cột (mỗi điểm dữ liệu) của \mathbf{X} đi trung bình các cột của nó. Ta gọi ma trận này $\widehat{\mathbf{X}}$ là *zero-corrected data* hoặc *dữ liệu đã được chuẩn hoá*. Có thể nhận thấy $\hat{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}}$, $\forall n = 1, 2, \dots, N$.

Vì vậy hàm mất mát trong (21.6) có thể được viết lại thành:

$$J = \frac{1}{N} \|\widehat{\mathbf{U}}_K^T \widehat{\mathbf{X}}\|_F^2 = \frac{1}{N} \|\widehat{\mathbf{X}}^T \widehat{\mathbf{U}}_K\|_F^2 = \frac{1}{N} \sum_{i=K+1}^D \|\widehat{\mathbf{X}}^T \mathbf{u}_i\|_2^2 \quad (21.8)$$

$$= \frac{1}{N} \sum_{i=K+1}^D \mathbf{u}_i^T \widehat{\mathbf{X}} \widehat{\mathbf{X}}^T \mathbf{u}_i = \sum_{i=K+1}^D \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i \quad (21.9)$$

với $\mathbf{S} = \frac{1}{N} \widehat{\mathbf{X}} \widehat{\mathbf{X}}^T$ là ma trận hiệp phương sai của dữ liệu và luôn là một ma trận nửa xác định dương (xem Mục 3.1.7).

Công việc còn lại là tìm các \mathbf{u}_i để mất mát là nhỏ nhất. Trước hết, chúng ta có một nhận xét thú vị. Với ma trận \mathbf{U} trực giao bất kỳ, thay $K = 0$ vào (21.9) ta có

$$L = \sum_{i=1}^D \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i = \frac{1}{N} \|\widehat{\mathbf{X}}^T \mathbf{U}\|_F^2 = \frac{1}{N} \text{trace}(\widehat{\mathbf{X}}^T \mathbf{U} \mathbf{U}^T \widehat{\mathbf{X}}) \quad (21.10)$$

$$= \frac{1}{N} \text{trace}(\widehat{\mathbf{X}}^T \widehat{\mathbf{X}}) = \frac{1}{N} \text{trace}(\widehat{\mathbf{X}} \widehat{\mathbf{X}}^T) = \text{trace}(\mathbf{S}) = \sum_{i=1}^D \lambda_i \quad (21.11)$$

Với $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D \geq 0$ là các trị riêng của ma trận nửa xác định dương \mathbf{S} . Chú ý rằng các trị riêng này là thực và không âm¹.

Như vậy L không phụ thuộc vào cách chọn ma trận trực giao \mathbf{U} và bằng tổng các phần tử trên đường chéo của \mathbf{S} . Nói cách khác, L chính là tổng của các phương sai theo từng thành phần của dữ liệu ban đầu².

Vì vậy, việc tối thiểu hàm mất mát J được cho bởi (21.9) tương đương với việc tối đa

$$F = L - J = \sum_{i=1}^K \mathbf{u}_i \mathbf{S} \mathbf{u}_i^T \quad (21.12)$$

21.1.3 Tối ưu hàm mất mát

Nghiệm của bài toán tối ưu hàm mất mát cho PCA được tìm dựa trên khẳng định sau đây.

Nếu \mathbf{S} là một ma trận nửa xác định dương, bài toán tối ưu

$$\max_{\mathbf{U}_K} \sum_{i=1}^K \mathbf{u}_i^T \mathbf{S} \mathbf{u}_i \quad (21.13)$$

$$\text{thỏa mãn: } \mathbf{U}_K^T \mathbf{U}_K = \mathbf{I} \quad (21.14)$$

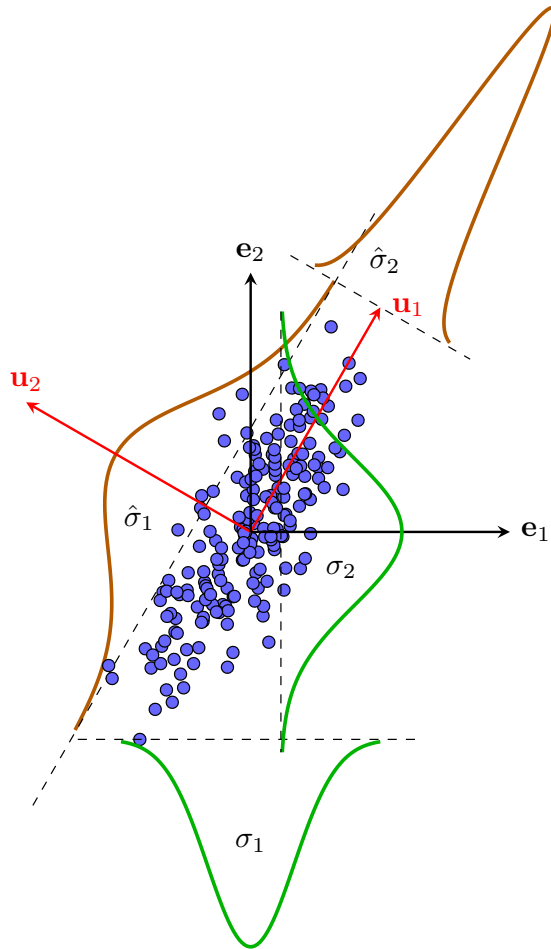
có nghiệm $\mathbf{u}_1, \dots, \mathbf{u}_K$ là các vector riêng ứng với K trị riêng (kể cả lặp) lớn nhất của \mathbf{S} . Khi đó, giá trị của hàm mục tiêu là $\sum_{i=1}^K \lambda_i$, với $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_D$ là các trị riêng của \mathbf{S} .

Khẳng định này có thể được chứng minh bằng quy nạp³.

¹ Tổng các trị riêng của một ma trận vuông bất kỳ luôn bằng trace của ma trận đó.

² Mỗi thành phần trên đường chéo chính của ma trận hiệp phương sai chính là phương sai của thành phần dữ liệu tương ứng.

³ Xin được bỏ qua phần chứng minh. Bạn đọc có thể xem Exercise 12.1 trong tài liệu tham khảo [Bis06] với lời giải tại <https://goo.gl/sM32pB>.



Hình 21.3: PCA có thể được coi là phương pháp đi tìm một hệ cơ sở trực chuẩn đóng vai trò một phép xoay, sao cho trong hệ cơ sở mới này, phương sai theo một số chiều nào đó là rất nhỏ, và ta có thể bỏ qua.

Trị riêng lớn nhất λ_1 của ma trận hiệp phương sai \mathbf{S} còn được gọi là *thành phần chính thứ nhất* (the first principal component), trị riêng thứ hai λ_2 còn được gọi là *thành phần chính thứ hai*, v.v.. Tên gọi *phân tích thành phần chính* (principal component analysis) bắt nguồn từ đây. Ta chỉ giữ lại K thành phần chính đầu tiên khi giảm chiều dữ liệu dùng PCA.

Hình 21.3 minh họa các thành phần chính với dữ liệu hai chiều. Trong không gian ban đầu với các vector cơ sở màu đen e_1, e_2 , phương sai theo mỗi chiều dữ liệu (độ rộng của các hình chuông màu lục) đều lớn. Trong không gian mới với các vector cơ sở màu đỏ u_1, u_2 , phương sai theo chiều thứ hai $\hat{\sigma}_2$ rất nhỏ so với $\hat{\sigma}_1$. Điều này nghĩa là khi chiếu dữ liệu lên u_2 ta được các điểm rất gần nhau và gần với giá trị trung bình theo chiều đó. Trong trường hợp này, giá trị trung bình theo mọi chiều bằng 0 nên ta có thể thay thế tọa độ theo chiều u_2 bằng 0. Rõ ràng là nếu dữ liệu có phương sai càng nhỏ theo một chiều nào đó thì khi xấp xỉ chiều đó bằng một hằng số, sai số xấp xỉ càng nhỏ. PCA thực chất là đi tìm một phép xoay tương ứng với một ma trận trực giao sao cho trong hệ tọa độ mới, tồn tại các chiều có phương sai nhỏ mà ta có thể bỏ qua; ta chỉ cần giữ lại các chiều/thành phần khác quan trọng hơn. Như đã khẳng định ở trên, tổng phương sai theo mọi chiều trong hệ cơ sở nào cũng là như nhau và bằng tổng các trị riêng của ma trận hiệp phương sai. Vì vậy, PCA còn được coi là phương pháp giảm số chiều dữ liệu sao tổng phương sai còn lại là lớn nhất.

21.2 Các bước thực hiện PCA

Từ các suy luận phía trên, ta có thể tóm tắt lại các bước trong PCA như sau:

1. Tính vector kỳ vọng của toàn bộ dữ liệu: $\bar{\mathbf{x}} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n$.
2. Trừ mỗi điểm dữ liệu đi vector kỳ vọng của toàn bộ dữ liệu để được dữ liệu chuẩn hoá:

$$\hat{\mathbf{x}}_n = \mathbf{x}_n - \bar{\mathbf{x}} \quad (21.15)$$

3. Đặt $\hat{\mathbf{X}} = [\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2, \dots, \hat{\mathbf{x}}_D]$ là ma trận dữ liệu chuẩn hoá, tính ma trận hiệp phương sai

$$\mathbf{S} = \frac{1}{N} \hat{\mathbf{X}} \hat{\mathbf{X}}^T \quad (21.16)$$

4. Tính các trị riêng và vector riêng tương ứng có ℓ_2 norm bằng 1 của ma trận này, sắp xếp chúng theo thứ tự giảm dần của trị riêng.
5. Chọn K vector riêng ứng với K trị riêng lớn nhất để xây dựng ma trận \mathbf{U}_K có các cột tạo thành một hệ trực giao. K vectors này, còn được gọi là các thành phần chính, tạo thành một không gian con *gần* với phân bố của dữ liệu ban đầu đã chuẩn hoá.
6. Chiếu dữ liệu ban đầu đã chuẩn hoá $\hat{\mathbf{X}}$ xuống không gian con tìm được.
7. Dữ liệu mới chính là tọa độ của các điểm dữ liệu trên không gian mới: $\mathbf{Z} = \mathbf{U}_K^T \hat{\mathbf{X}}$.

Dữ liệu ban đầu có thể tính được xấp xỉ theo dữ liệu mới bởi $\mathbf{x} \approx \mathbf{U}_K \mathbf{Z} + \bar{\mathbf{x}}$.

Một điểm dữ liệu mới $\mathbf{v} \in \mathbb{R}^D$ (có thể không nằm trong tập huấn luyện) sẽ được giảm chiều bằng PCA theo công thức $\mathbf{w} = \mathbf{U}_K^T (\mathbf{v} - \bar{\mathbf{x}}) \in \mathbb{R}^K$. Ngược lại, nếu biết \mathbf{w} , ta có thể xấp xỉ \mathbf{v} bởi $\mathbf{U}_K \mathbf{w} + \bar{\mathbf{x}}$. Các bước thực hiện PCA được minh hoạ trong Hình 21.4.

21.3 Mối quan hệ giữa PCA và SVD

Giữa PCA và SVD có mối quan hệ đặc biệt với nhau. Để nhận ra điều này, tôi xin được nhắc lại hai điểm đã trình bày sau đây:

21.3.1 SVD cho bài toán xấp xỉ low-rank tốt nhất

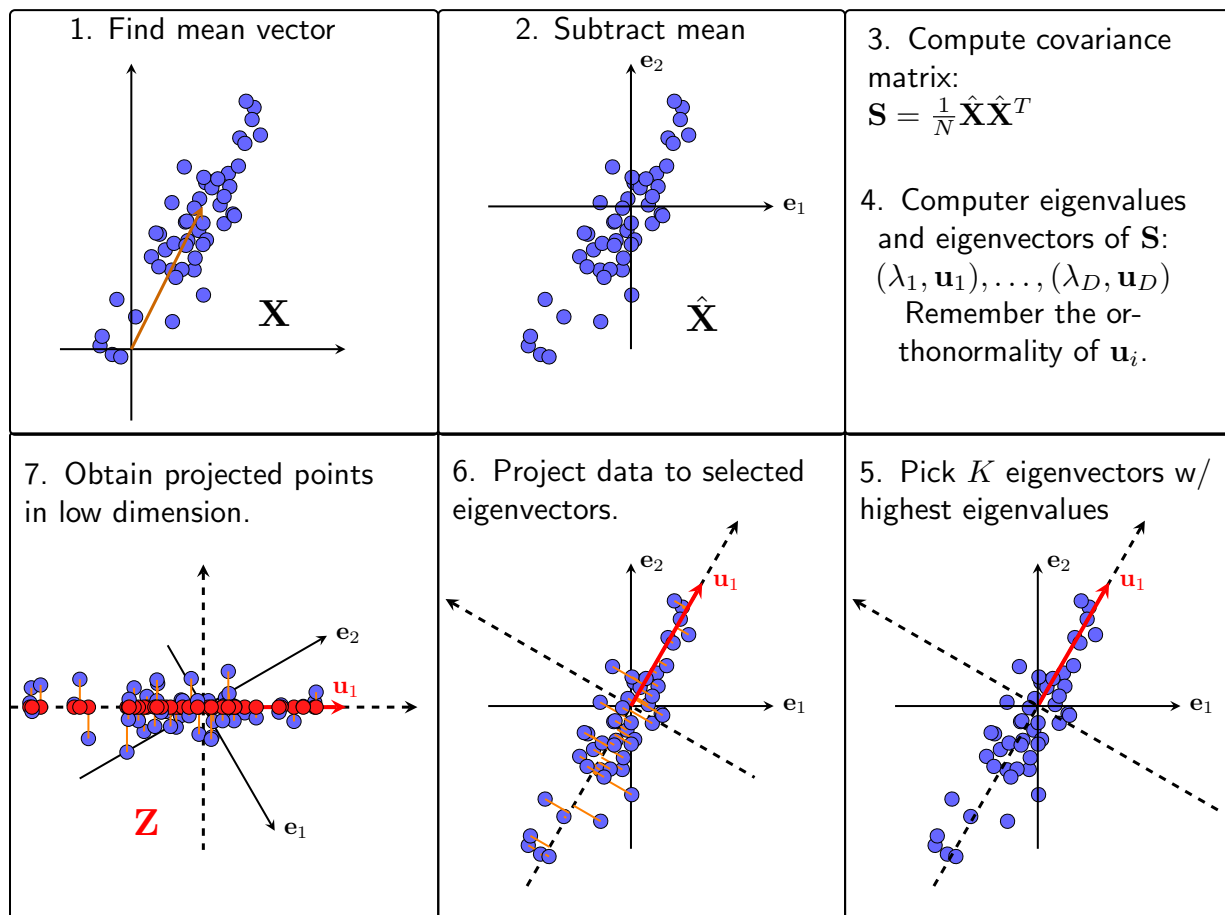
Nghiệm \mathbf{A} của bài toán xấp xỉ một ma trận bởi một ma trận có rank không vượt quá k :

$$\min_{\mathbf{A}} \|\mathbf{X} - \mathbf{A}\|_F \quad (21.17)$$

thoả mãn: $\text{rank}(\mathbf{A}) = K$

chính là truncated SVD của \mathbf{A} .

PCA procedure



Hình 21.4: Các bước thực hiện PCA.

Cụ thể, nếu SVD của $\mathbf{X} \in \mathbb{R}^{D \times N}$ là

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{21.18}$$

với $\mathbf{U} \in \mathbb{R}^{D \times D}$ và $\mathbf{V} \in \mathbb{R}^{N \times N}$ là các ma trận trực giao, và $\mathbf{\Sigma} \in \mathbb{R}^{D \times N}$ là ma trận đường chéo (không nhất thiết vuông) với các phần tử trên đường chéo không âm giảm dần. Nghiệm của bài toán (21.17) chính là:

$$\mathbf{A} = \mathbf{U}_K \mathbf{\Sigma}_K \mathbf{V}_K^T \tag{21.19}$$

với $\mathbf{U} \in \mathbb{R}^{D \times K}$ và $\mathbf{V} \in \mathbb{R}^{N \times K}$ là các ma trận tạo bởi K cột đầu tiên của \mathbf{U} và \mathbf{V} , và $\mathbf{\Sigma}_K \in \mathbb{R}^{K \times K}$ là ma trận đường chéo con ứng với K hàng đầu tiên và K cột đầu tiên của $\mathbf{\Sigma}$.

21.3.2 Ý tưởng của PCA

Trong PCA, như đã chứng minh ở (21.5), PCA là bài toán đi tìm ma trận trực giao \mathbf{U} và ma trận mô tả dữ liệu ở không gian thấp chiều \mathbf{Z} sao cho việc xấp xỉ sau đây là tốt nhất:

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_K \mathbf{Z} + \hat{\mathbf{U}}_K \hat{\mathbf{U}}_K^T \bar{\mathbf{x}} \mathbf{1}^T \tag{21.20}$$

với $\mathbf{U}_K, \widehat{\mathbf{U}}_K$ lần lượt là các ma trận được tạo bởi K cột đầu tiên và $D - K$ cột cuối cùng của ma trận trực giao \mathbf{U} , và $\bar{\mathbf{x}}$ là vector kỳ vọng của dữ liệu.

Giả sử rằng vector kỳ vọng $\bar{\mathbf{x}} = \mathbf{0}$. Khi đó, (21.20) tương đương với

$$\mathbf{X} \approx \tilde{\mathbf{X}} = \mathbf{U}_K \mathbf{Z} \quad (21.21)$$

Bài toán tối ưu của PCA sẽ trở thành:

$$\begin{aligned} \mathbf{U}_K, \mathbf{Z} = \arg \min_{\mathbf{U}_K, \mathbf{Z}} \|\mathbf{X} - \mathbf{U}_K \mathbf{Z}\|_F \\ \text{thoả mãn:} \quad \mathbf{U}_K^T \mathbf{U}_K = \mathbf{I}_K \end{aligned} \quad (21.22)$$

với $\mathbf{I}_K \in \mathbb{R}^{K \times K}$ là ma trận đơn vị trong không gian K chiều, và điều kiện ràng buộc là để đảm bảo các cột của \mathbf{U}_K tạo thành một hệ trực chuẩn.

21.3.3 Quan hệ giữa PCA và SVD

Bạn có nhận ra điểm tương đồng giữa hai bài toán tối ưu (21.17) và (21.22) với nghiệm của bài toán đầu tiên được cho trong (21.19)? Bạn có thể nhận ra ngay nghiệm của bài toán (21.22) chính là

$$\begin{aligned} \mathbf{U}_K \text{ trong (21.22)} &= \mathbf{U}_K \text{ trong (21.19)} \\ \mathbf{Z} \text{ trong (21.22)} &= \Sigma_K \mathbf{V}_K^T \text{ trong (21.19)} \end{aligned}$$

Vậy, nếu các điểm dữ liệu được biểu diễn bởi các cột của một ma trận, và trung bình cộng của mỗi hàng của ma trận đó bằng 0 (để cho vector trung bình bằng 0), thì nghiệm của bài toán PCA được rút ra trực tiếp từ truncated SVD của ma trận đó. Nói cách khác, việc đi tìm nghiệm cho PCA chính là việc giải một bài toán matrix factorization thông qua SVD.

21.4 Làm thế nào để chọn số chiều của dữ liệu mới

Một câu hỏi được đặt ra là, làm thế nào để chọn ra giá trị K – chiều của dữ liệu mới – với từng dữ liệu cụ thể?

Có một cách xác định K là dựa trên việc *lượng thông tin muốn giữ lại*. Như đã trình bày, PCA còn được gọi là phương pháp tối đa *tổng phương sai được giữ lại*. Vậy ta có thể coi tổng các phương sai được giữ lại là lượng thông tin được giữ lại.

Nhắc lại rằng trong mọi hệ trục tọa độ, tổng phương sai của dữ liệu là như nhau và bằng tổng các trị riêng của ma trận hiệp phương sai $\sum_{i=1}^D \lambda_i$. Thêm nữa, PCA giúp giữ lại lượng thông tin (tổng các phương sai) là $\sum_{i=1}^K \lambda_i$. Vậy ta có thể coi biểu thức:

$$r_K = \frac{\sum_{i=1}^K \lambda_i}{\sum_{j=1}^D \lambda_j} \quad (21.23)$$

là tỉ lệ thông tin được giữ lại khi số chiều dữ liệu mới sau PCA là K . Như vậy, giả sử ta muốn giữ lại 99% dữ liệu, ta chỉ cần chọn K là số tự nhiên nhỏ nhất sao cho $r_K \geq 0.99$.

Khi dữ liệu phân bố quanh một không gian con, các giá trị phương sai lớn nhất ứng với các λ_i đầu tiên lớn hơn nhiều so với các phương sai còn lại. Khi đó, ta có thể chọn được K khá nhỏ để đạt được $r_K \geq 0.99$.

21.5 Lưu ý về tính PCA trong các bài toán thực tế

Có hai trường hợp trong thực tế mà chúng ta cần lưu ý về PCA. Trường hợp thứ nhất là lượng dữ liệu có được nhỏ hơn rất nhiều so với số chiều dữ liệu. Trường hợp thứ hai là khi lượng dữ liệu trong tập huấn luyện là rất lớn, có thể lên tới cả triệu. Việc tính toán ma trận hiệp phương sai và trị riêng đôi khi trở nên bất khả thi. Có những hướng giải quyết hiệu quả cho các trường hợp này.

Trong mục này, ta sẽ coi như dữ liệu đã được chuẩn hoá, tức đã được trừ đi vector kỳ vọng. Khi đó, ma trận hiệp phương sai sẽ là $\mathbf{S} = \frac{1}{N}\mathbf{X}\mathbf{X}^T$.

21.5.1 Số chiều dữ liệu nhiều hơn số điểm dữ liệu

Đó là trường hợp $D > N$, tức ma trận dữ liệu \mathbf{X} là một *ma trận cao*. Khi đó, số trị riêng khác không của ma trận hiệp phương sai \mathbf{S} sẽ không vượt quá rank của nó, tức không vượt quá N . Vậy ta cần chọn $K \leq N$ vì không thể chọn ra được nhiều hơn N trị riêng khác 0 của một ma trận có rank bằng N .

Việc tính toán các trị riêng và vector riêng cũng có thể được thực hiện một cách hiệu quả dựa trên các tính chất sau đây:

Tính chất 1: Trị riêng của \mathbf{A} cũng là trị riêng của $k\mathbf{A}$ với $k \neq 0$ bất kỳ. Điều này có thể được suy ra trực tiếp từ định nghĩa của trị riêng và vector riêng.

Tính chất 2: Trị riêng của \mathbf{AB} cũng là trị riêng của \mathbf{BA} với $\mathbf{A} \in \mathbb{R}^{d_1 \times d_2}$, $\mathbf{B} \in \mathbb{R}^{d_2 \times d_1}$ là các ma trận bất kỳ và d_1, d_2 là các số tự nhiên khác không bất kỳ.

Như vậy, thay vì tìm trị riêng của ma trận hiệp phương sai $\mathbf{S} \in \mathbb{R}^{D \times D}$, ta đi tìm trị riêng của ma trận $\mathbf{T} = \mathbf{X}^T\mathbf{X} \in \mathbb{R}^{N \times N}$ có số chiều nhỏ hơn (vì $N < D$).

Tính chất 3: Giả sử (λ, \mathbf{u}) là một cặp trị riêng - vector riêng của \mathbf{T} , thế thì $(\lambda, \mathbf{X}\mathbf{u})$ là một cặp trị riêng - vector riêng của \mathbf{S} . Thật vậy,

$$\mathbf{X}^T\mathbf{X}\mathbf{u} = \mathbf{T}\mathbf{u} = \lambda\mathbf{u} \Rightarrow (\mathbf{X}\mathbf{X}^T)(\mathbf{X}\mathbf{u}) = \lambda(\mathbf{X}\mathbf{u}) \quad (21.24)$$

Dấu bằng thứ nhất xảy ra theo định nghĩa của trị riêng và vector riêng. Từ (21.24) ta suy ra **Tính chất 3**.

Như vậy, ta có thể hoàn toàn tính được trị riêng và vector riêng của ma trận hiệp phương sai \mathbf{S} dựa trên một ma trận \mathbf{T} có kích thước nhỏ hơn. Việc này trong nhiều trường hợp khiến thời gian tính toán giảm đi đáng kể.

21.5.2 Với các bài toán large-scale

Trong rất nhiều bài toán, cả D và N đều là các số rất lớn, đồng nghĩa với việc ta phải tìm trị riêng cho một ma trận rất lớn. Ví dụ, có một triệu bức ảnh 1000×1000 pixel, như vậy $D = N = 10^6$ là một số rất lớn, việc trực tiếp tính toán trị riêng và vector riêng cho ma trận hiệp phương sai là không khả thi. Tuy nhiên, có một phương pháp cho phép tính xấp xỉ các giá trị này một cách nhanh hơn. Phương pháp đó có tên là *power method* (<https://goo.gl/eBRPxH>).

21.6 Một vài ứng dụng của PCA

Ứng dụng đầu tiên có thể thấy của PCA chính là việc giảm chiều dữ liệu, giúp việc lưu trữ và tính toán được thuận tiện hơn. Thực tế cho thấy, nhiều khi làm việc trên dữ liệu đã được giảm chiều mang lại kết quả tốt hơn cho với dữ liệu gốc. Thứ nhất, có thể phần dữ liệu mang thông tin nhỏ bị lược đi chính là phần gây nhiễu, những thông tin quan trọng hơn đã được giữ lại. Thứ hai, số điểm dữ liệu nhiều khi ít hơn số chiều dữ liệu. Khi có quá ít dữ liệu và số chiều dữ liệu quá lớn, overfitting rất dễ xảy ra. Việc giảm chiều dữ liệu phần nào giúp khắc phục hiện tượng này.

Dưới đây là hai ví dụ về ứng dụng của PCA trong bài toán *face classification* và *anomaly detection* (dò điểm bất thường).

21.6.1 Eigenface

Eigenface từng là một trong các kỹ thuật phổ biến nhất trong bài toán nhận dạng khuôn mặt. Giả sử rằng vị trí các khuôn mặt đã được xác định trong ảnh và có kích thước như nhau, bài toán đặt ra là xác định đó là khuôn mặt của ai. Ý tưởng của eigenface là đi tìm một không gian có số chiều nhỏ hơn để mô tả mỗi khuôn mặt, từ đó sử dụng vector trong không gian thấp này như là vector đặc trưng đưa vào các bộ phân lớp. Điều đáng nói là một bức ảnh khuôn mặt có kích thước khoảng 200×200 sẽ có số chiều là 40k – là một số rất lớn, trong khi đó, vector đặc trưng thường chỉ có số chiều bằng vài trăm hoặc vài nghìn. Eigenface thực ra chính là PCA. Các eigenface chính là các vector riêng (eigenvector) ứng với các trị riêng lớn nhất của ma trận hiệp phương sai.

Trong phần này, chúng ta cùng làm một thí nghiệm nhỏ trên cơ sở dữ liệu Yale face database (<https://goo.gl/LNg8LS>). Các bức ảnh trong thí nghiệm này đã được căn chỉnh cho cùng với kích thước và khuôn mặt nằm trọn vẹn trong một hình chữ nhật có kích thước 116×98 pixel. Có tất cả 15 người khác nhau, mỗi người có 11 bức ảnh được chụp ở các điều kiện ánh sáng và cảm xúc khác nhau, bao gồm 'centerlight', 'glasses', 'happy', 'leftlight', 'noglases', 'normal', 'rightlight', 'sad', 'sleepy', 'surprised', và 'wink'. Hình 21.5 minh họa các bức ảnh của người có id là 10.

Ta có thể thấy rằng số chiều dữ liệu là $116 \times 98 = 11368$ là một số khá lớn. Tuy nhiên, vì chỉ có tổng cộng $15 \times 11 = 165$ bức ảnh nên ta có thể nén các bức ảnh này về dữ liệu mới có chiều nhỏ hơn 165. Trong ví dụ này, chúng ta chọn $K = 100$.



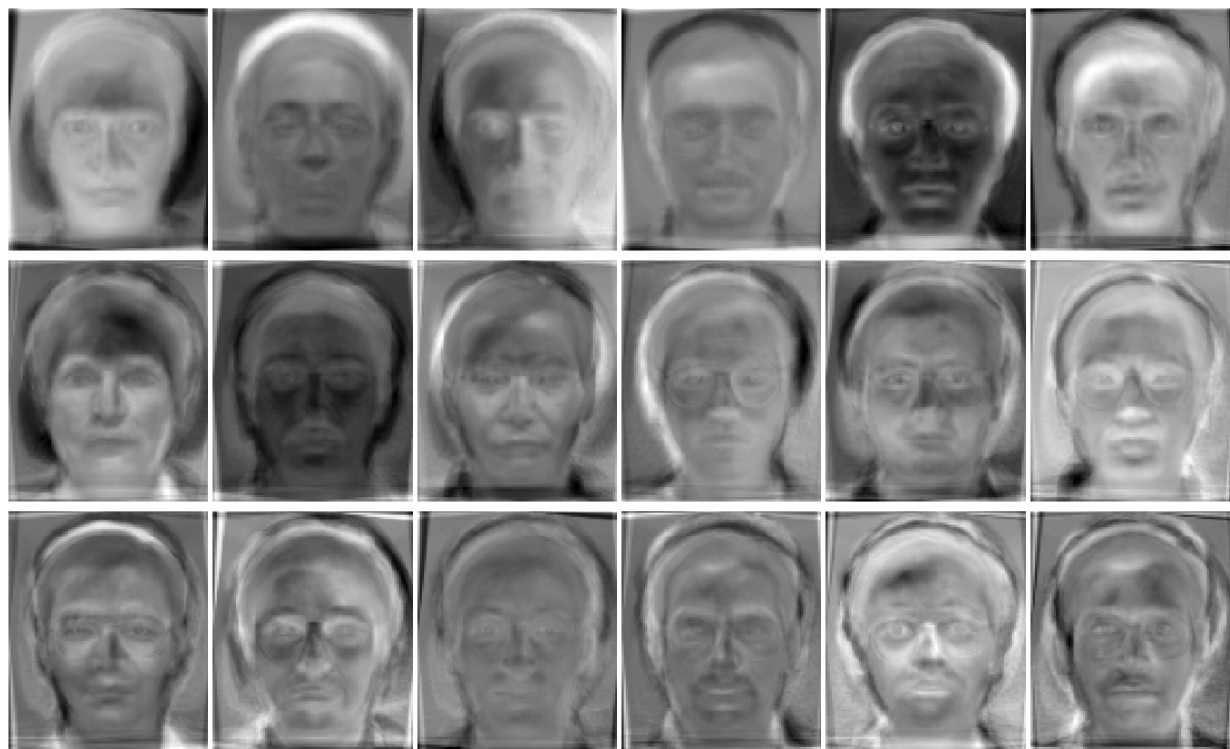
Hình 21.5: Ví dụ về ảnh của một người trong Yale Face Database.

Dưới đây là đoạn code thực hiện PCA cho toàn bộ dữ liệu. Ở đây chúng ta trực tiếp sử dụng PCA trong **sklearn**.

```
import numpy as np
from scipy import misc # for loading image
np.random.seed(1)

# filename structure
path = 'unpadded/' # path to the database
ids = range(1, 16) # 15 persons
states = ['centerlight', 'glasses', 'happy', 'leftlight',
          'noglasses', 'normal', 'rightlight', 'sad',
          'sleepy', 'surprised', 'wink' ]
prefix = 'subject'
surfix = '.pgm'
# data dimension
h, w, K = 116, 98, 100 # hight, weight, new dim
D = h * w
N = len(states)*15
# collect all data
X = np.zeros((D, N))
cnt = 0
for person_id in range(1, 16):
    for state in states:
        fn = path + prefix + str(person_id).zfill(2) + '.' + state + surfix
        X[:, cnt] = misc.imread(fn).reshape(D)
        cnt += 1

# Doing PCA, note that each row is a datapoint
from sklearn.decomposition import PCA
pca = PCA(n_components=K) # K = 100
pca.fit(X.T)
# projection matrix
U = pca.components_.T
```

Hình 21.6: Các eigenfaces tìm được bằng PCA.

Trong dòng `pca = PCA(n_components=K)`, nếu `n_components` là một số thực trong khoảng $(0, 1)$, PCA sẽ thực hiện việc tìm K dựa trên biểu thức (21.23).

Hình 21.6 biểu diễn 18 vector riêng đầu tiên (18 cột đầu tiên của U_k) tìm được bằng PCA. Các vector đã được `reshape` về cùng kích thước như các bức ảnh gốc. Có một điều dễ nhận ra là các ảnh minh họa các vector thu được ít nhiều mang thông tin của mặt người. Thực tế, một khuôn mặt gốc sẽ được xấp xỉ như tổng có trọng số của các *khuôn mặt* này. Vì các vector riêng này đóng vai trò như cơ sở của không gian mới với ít chiều hơn, chúng còn được gọi là *khuôn mặt chính*, tức *eigenface*⁴. Để xem mức độ hiệu quả của Eigenface, chúng ta thử minh họa các bức ảnh gốc và các bức ảnh được xấp xỉ bằng PCA, kết quả được cho như trên Hình 21.7. Các khuôn mặt nhận được vẫn mang khá đầy đủ thông tin của các khuôn mặt gốc. Điều đang nói hơn, các khuôn mặt trong hàng dưới được suy ra từ một vector 100 chiều, so với 11368 chiều như ở hàng trên.

Source code cho chương này có thể được tìm thấy tại <https://goo.gl/zQ3DSZ>.

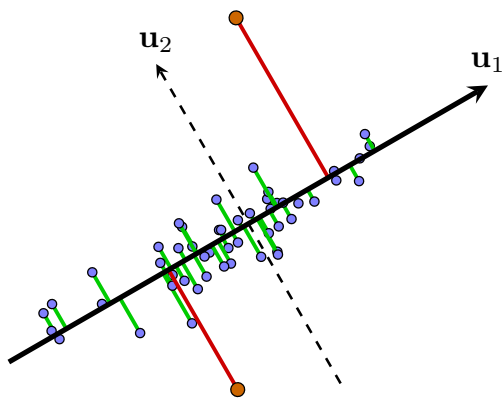
21.6.2 Abnormal Detection

Ngoài các ứng dụng về nén và phân lớp, PCA còn được sử dụng trong nhiều lĩnh vực khác nhau. *Abnormal detection*, hay *outlier detection* (xác định các hiện tượng không bình thường)

⁴ Từ *riêng* trong trường hợp này không thực sự truyền tải thông tin. Từ *chính* được dùng vì nó đi kèm với văn cảnh của *phân tích thành phần chính*.



Hình 21.7: Hàng trên: các ảnh gốc. Hàng dưới: các ảnh được suy ra từ eigenfaces. Ảnh ở hàng dưới có nhiều nhiễu nhưng vẫn mang những đặc điểm riêng mà mắt người có thể phân biệt được.



Hình 21.8: PCA cho việc xác định các sự kiện *bất thường*. Giả sử rằng các sự kiện *bình thường* chiếm đa số và nằm gần trong một không gian con nào đó. Khi đó, nếu làm PCA trên toàn bộ dữ liệu, không gian con thu được gần với không gian con của tập các sự kiện *bình thường*. Lúc này, các điểm quá xa không gian con này, trong trường hợp này là các điểm màu cam, có thể được coi là các sự kiện *bất thường*.

là một trong số đó [SCSC03, LCD04]. Thêm nữa, giả sử chúng ta không biết nhãn của các sự kiện này, tức ta đang làm việc với một bài toán không giám sát.

Ý tưởng cơ bản là các sự kiện bình thường có thể nằm gần một không gian con nào đó, trong khi các sự kiện bất thường thường nằm xa không gian con đó. Hơn nữa, vì là bất thường nên số lượng các sự kiện thuộc loại này là rất nhỏ so với các sự kiện bình thường. Như vậy, chúng ta có thể làm PCA trên toàn bộ dữ liệu để tìm ra các thành phần chính của dữ liệu, từ đó suy ra không gian con mà các điểm bình thường nằm gần. Việc xác định một điểm là bình thường hay bất thường được xác định bằng cách đo khoảng cách từ điểm đó tới không gian con tìm được. Hình 21.8 minh họa cho việc xác định các sự kiện bất thường bằng PCA.

21.7 Thảo luận

- PCA là một phương pháp giảm chiều dữ liệu dựa trên việc tối đa lượng thông tin được giữ lại. Lượng thông tin được giữ lại được đo bằng tổng các phương sai trên mỗi thành

phần của dữ liệu. Lượng dữ liệu sẽ được giữ lại nhiều nhất khi các chiều dữ liệu còn lại tương ứng với các vector riêng của trị riêng lớn nhất của ma trận hiệp phương sai.

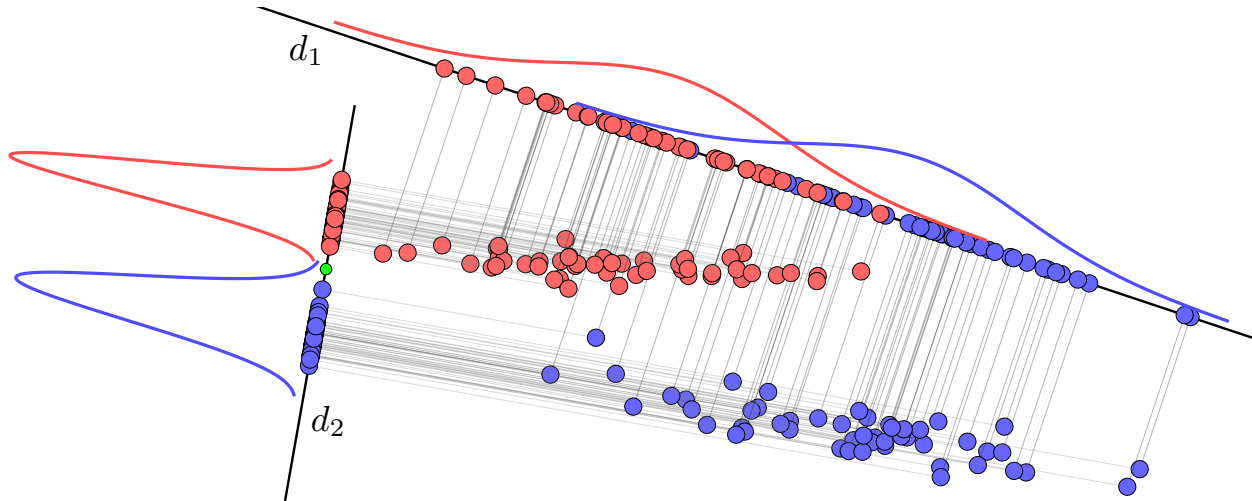
- Với các bài toán large-scale, đôi khi việc tính toán trên toàn bộ dữ liệu là không khả thi vì còn có vấn đề về bộ nhớ. Giải pháp là thực hiện PCA lần đầu trên một tập con dữ liệu vừa với bộ nhớ, sau đó lấy một tập con khác để *từ từ (incrementally)* cập nhật nghiệm của PCA tới khi nào hội tụ. Ý tưởng này khá giống với mini-batch Gradient Descent, và được gọi là Incremental PCA [ZYK06].
- Ngoài ra, còn rất nhiều hướng mở rộng của PCA, bạn đọc có thể tìm kiếm theo từ khoá: Sparse PCA [dGJL05], Kernel PCA [MSS+99], Robust PCA [CLMW11].

Linear discriminant analysis

22.1 Giới thiệu

Trong chương trước, chúng ta đã làm quen với một thuật toán giảm chiều dữ liệu phổ biến nhất – principle component analysis (PCA). Như đã đề cập, PCA là một mô hình unsupervised learning, tức là nó chỉ sử dụng các vector mô tả dữ liệu mà không cần tới nhãn, nếu có, của dữ liệu. Tuy nhiên, trong bài toán phân lớp, việc khai thác mối liên quan giữa dữ liệu và nhãn sẽ mang lại kết quả phân loại tốt hơn.

Nhắc lại rằng PCA là phương pháp giảm chiều dữ liệu sao cho lượng thông tin về dữ liệu, thể hiện ở tổng phương sai của các thành phần được giữ lại, được giữ lại là nhiều nhất. Tuy nhiên, trong nhiều bài toán, ta không cần giữ lại lượng thông tin lớn nhất mà chỉ cần giữ lại thông tin cần thiết cho riêng bài toán đó. Xét ví dụ về bài toán phân lớp nhị phân được mô tả trong Hình 22.1. Ở đây, ta giả sử rằng dữ liệu được chiếu lên một đường thẳng và mỗi điểm được thay bởi hình chiếu của nó lên đường thẳng kia. Như vậy, số chiều dữ liệu đã được giảm từ hai về một. Câu hỏi đặt ra là, đường thẳng cần có phương như thế nào để hình chiếu của dữ liệu trên đường thẳng này *giúp ích cho việc phân lớp nhất*? Việc phân lớp đơn giản nhất có thể được hiểu là việc tìm ra một ngưỡng giúp phân tách hai lớp một cách đơn giản và đạt kết quả tốt nhất. Xét hai đường thẳng d_1 và d_2 . Trong đó phương của d_1 gần với phương của thành phần chính nếu thực hiện PCA, phương của d_2 gần với phương của thành phần phụ tìm được bằng PCA. Nếu ta làm giảm chiều dữ liệu bằng PCA, ta sẽ thu được dữ liệu gần với các điểm được chiếu lên d_1 . Lúc này việc phân tách hai lớp trở nên phức tạp vì các điểm dữ liệu mới của hai lớp chồng lấn lên nhau. Ngược lại, nếu ta chiếu dữ liệu lên đường thẳng gần với thành phần phụ tìm được bởi PCA, tức d_2 , các điểm hình chiếu nằm hoàn toàn về hai phía khác nhau của điểm màu lục trên đường thẳng này. Với bài toán phân lớp, việc chiếu dữ liệu lên d_2 vì vậy sẽ mang lại hiệu quả hơn. Việc phân loại một điểm dữ liệu mới sẽ được xác định nhanh chóng bằng cách so sánh hình chiếu của nó lên d_2 với điểm màu xanh lục này.

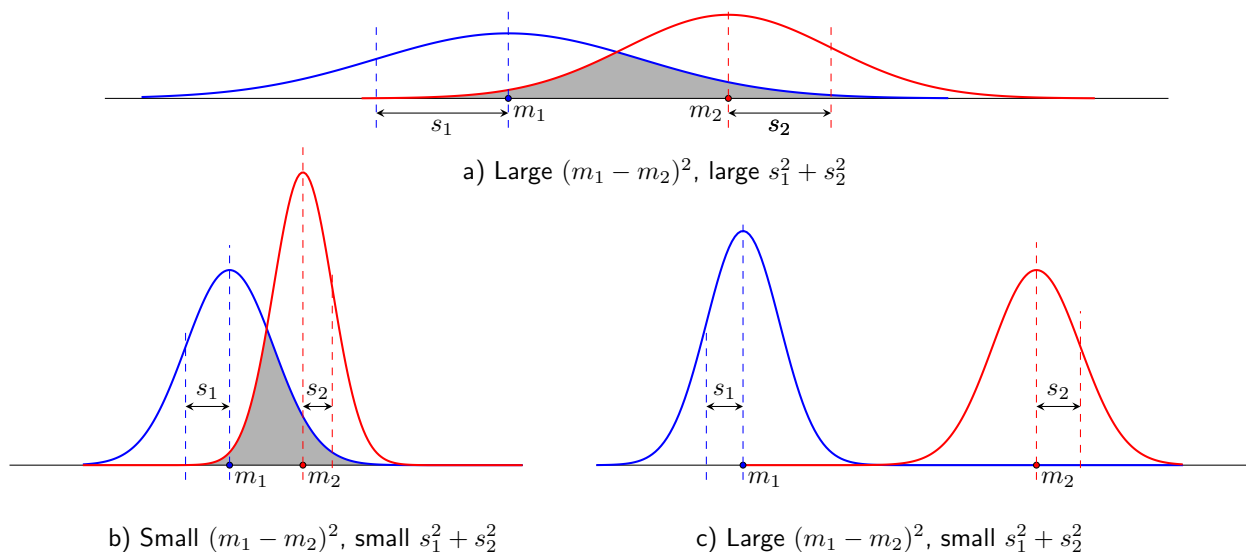


Hình 22.1: Chiếu dữ liệu lên các đường thẳng khác nhau. Có hai lớp dữ liệu minh họa bởi các điểm màu xanh và đỏ trong không gian hai chiều. Số chiều được giảm về một bằng cách chiếu dữ liệu lên các đường thẳng khác nhau d_1 và d_2 . Trong hai cách chiếu này, phương của d_1 gần giống với phương của thành phần chính thứ nhất của dữ liệu, phương của d_2 gần với thành phần phụ của dữ liệu nếu dùng PCA. Khi chiếu dữ liệu lên d_1 , nhiều điểm màu đỏ và xanh bị chồng lấn lên nhau, khiến cho việc phân loại dữ liệu là không khả thi trên đường thẳng này. Ngược lại, khi được chiếu lên d_2 , dữ liệu của hai lớp được chia thành các cụm tương ứng tách biệt nhau, khiến cho việc phân lớp trở nên đơn giản hơn và hiệu quả hơn. Các đường cong hình chuông thể hiện xấp xỉ phân bố xác suất của dữ liệu hình chiếu trong mỗi lớp.

Qua ví dụ trên ta thấy rằng, **không phải trong mọi trường hợp việc giữ lại thông tin nhiều nhất sẽ luôn mang lại kết quả tốt nhất**. Chú ý rằng kết quả của phân tích trên đây không có nghĩa là thành phần phụ mang lại hiệu quả tốt hơn thành phần chính. Việc chiếu dữ liệu lên đường thẳng nào giúp ích cho các bài toán phân lớp cần nhiều phân tích cụ thể hơn nữa. Ngoài ra, hai đường thẳng d_1 và d_2 trên đây không vuông góc với nhau, chúng được chọn gần với các thành phần chính và phụ của dữ liệu phục vụ cho mục đích minh họa.

Linear discriminant analysis (LDA) được ra đời nhằm tìm ra phương chiếu dữ liệu hiệu quả cho bài toán phân lớp. LDA có thể được coi là một phương pháp giảm chiều dữ liệu, cũng có thể được coi là một phương pháp phân lớp, và cũng có thể được áp dụng đồng thời cho cả hai, tức giảm chiều dữ liệu sao cho việc phân lớp hiệu quả nhất. Số chiều của dữ liệu mới là nhỏ hơn hoặc bằng $C - 1$ trong đó C là số lượng lớp dữ liệu. Từ *discriminant* được hiểu là *những thông tin đặc trưng cho mỗi lớp, khiến nó không bị lẫn với các classes khác*. Từ *linear* được dùng vì cách giảm chiều dữ liệu được thực hiện dựa trên một ma trận chiếu – là một phép biến đổi tuyến tính.

Trong Mục 22.2 dưới đây, chúng ta sẽ thảo luận về LDA cho bài toán phân lớp nhị phân. Mục 22.3 sẽ tổng quát LDA lên cho trường hợp với nhiều lớp dữ liệu. Mục 22.4 sẽ có các ví dụ và code Python cho LDA.



Hình 22.2: Khoảng cách giữa các kỳ vọng và tổng các phương sai ảnh hưởng tới độ *discriminant* của dữ liệu. (a) Khoảng cách giữa hai kỳ vọng là lớn nhưng phương sai trong mỗi class cũng lớn, khiến cho hai phân phối chồng lấn lên nhau (phần màu xám). (b) Phương sai cho mỗi class là rất nhỏ nhưng hai kỳ vọng quá gần nhau, khiến khó phân biệt hai class. (c) Khi phương sai đủ nhỏ và khoảng cách giữa hai kỳ vọng đủ lớn, ta thấy rằng dữ liệu *discriminative* hơn.

22.2 LDA cho bài toán phân lớp nhị phân

22.2.1 Ý tưởng cơ bản

Quay lại với Hình 22.1, giả sử rằng dữ liệu của mỗi lớp khi được chiếu xuống một đường thẳng tuân theo phân phối chuẩn, có hàm mật độ xác suất là một đường hình chuông. Độ rộng của mỗi đường hình chuông này thể hiện *độ lệch chuẩn* (*standard deviation*¹, ký hiệu là s) của dữ liệu. Dữ liệu càng tập trung thì độ lệch chuẩn càng nhỏ, càng phân tán thì độ lệch chuẩn càng cao. Khi được chiếu lên d_1 , dữ liệu của hai lớp bị phân tán quá nhiều, khiến cho chúng bị trộn lẫn vào nhau. Khi được chiếu lên d_2 , mỗi lớp đều có độ lệch chuẩn nhỏ, khiến cho dữ liệu trong từng lớp tập trung hơn, dẫn đến kết quả tốt hơn.

Tuy nhiên, việc độ lệch chuẩn nhỏ trong mỗi lớp chưa đủ để đảm bảo độ *discriminant* của dữ liệu là tốt hơn. Xét các ví dụ trong Hình 22.2. Hình 22.2a chính là trường hợp trong Hình 22.1 khi dữ liệu được chiếu lên d_1 . Cả hai lớp đều quá phân tán khiến cho lượng chồng lấn (phần diện tích màu xám) là lớn, tức dữ liệu chưa thực sự *discriminative*. Hình 22.2b là trường hợp khi độ lệch chuẩn của hai lớp đều nhỏ, tức dữ liệu trong mỗi lớp tập trung hơn. Tuy nhiên, vấn đề với trường hợp này là khoảng cách giữa hai lớp, được đo bằng khoảng cách giữa hai kỳ vọng m_1 và m_2 , là quá nhỏ, khiến cho phần chồng lấn cũng chiếm một tỉ lệ lớn, và tất nhiên, cũng không tốt cho việc phân lớp. Hình 22.2c là trường hợp khi cả hai độ lệch chuẩn là nhỏ và khoảng cách giữa hai kỳ vọng là lớn, phần chồng lấn nhỏ không đáng kể.

¹ độ lệch chuẩn là căn bậc hai của phương sai

Vậy, độ lệch chuẩn và khoảng cách giữa hai kỳ vọng cụ thể đại diện cho các tiêu chí gì?

- Độ lệch chuẩn nhỏ thể hiện việc dữ liệu ít phân tán, tức dữ liệu trong mỗi lớp có xu hướng giống nhau. Hai phương sai s_1^2, s_2^2 còn được gọi là các **within-class variance**.
- Khoảng cách giữa các kỳ vọng là lớn chứng tỏ rằng hai lớp nằm xa nhau, tức dữ liệu giữa các lớp là khác nhau nhiều. Bình phương khoảng cách giữa hai kỳ vọng $(m_1 - m_2)^2$ còn được gọi là **between-class variance**.

Hai lớp dữ liệu được gọi là *discriminative* nếu hai lớp đó cách xa nhau (between-class variance lớn) và dữ liệu trong mỗi lớp có xu hướng giống nhau (within-class variance nhỏ). LDA là thuật toán đi tìm một phép chiếu sao cho tỉ lệ giữa *between-class variance* và *within-class variance* lớn nhất có thể.

22.2.2 Hàm mục tiêu của LDA

Giả sử rằng có N điểm dữ liệu $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N \in \mathbb{R}^D$ trong đó $N_1 < N$ điểm đầu tiên thuộc lớp thứ nhất, $N_2 = N - N_1$ điểm còn lại thuộc lớp thứ hai. Ký hiệu $\mathcal{C}_1 = \{n | 1 \leq n \leq N_1\}$ là tập hợp các chỉ số của các điểm thuộc lớp thứ nhất và $\mathcal{C}_2 = \{m | N_1 + 1 \leq m \leq N\}$ là tập hợp các chỉ số của các điểm thuộc lớp thứ hai. Phép chiếu dữ liệu xuống một đường thẳng có thể được mô tả bằng một vector hệ số \mathbf{w} , giá trị tương ứng của mỗi điểm dữ liệu mới được cho bởi

$$z_n = \mathbf{w}^T \mathbf{x}_n, 1 \leq n \leq N \quad (22.1)$$

Vector kỳ vọng của mỗi lớp được tính bởi

$$\mathbf{m}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{x}_n, \quad k = 1, 2 \quad (22.2)$$

$$\Rightarrow m_1 - m_2 = \frac{1}{N_1} \sum_{i \in \mathcal{C}_1} z_i - \frac{1}{N_2} \sum_{j \in \mathcal{C}_2} z_j = \mathbf{w}^T (\mathbf{m}_1 - \mathbf{m}_2) \quad (22.3)$$

Các **within-class variance** được định nghĩa là

$$s_k^2 = \sum_{n \in \mathcal{C}_k} (z_n - m_k)^2, \quad k = 1, 2 \quad (22.4)$$

Chú ý rằng các within-class variance ở đây không được lấy trung bình như phương sai thông thường. Điều này được lý giải là tầm quan trọng của mỗi within-class variance nên tỉ lệ thuận với số lượng điểm dữ liệu trong lớp đó, tức within-class variance bằng phương sai nhân với số điểm trong lớp đó.

LDA là thuật toán đi tìm giá trị lớn nhất của hàm mục tiêu

$$J(\mathbf{w}) = \frac{(m_1 - m_2)^2}{s_1^2 + s_2^2} \quad (22.5)$$

Thông qua việc tối đa hàm mục tiêu này, ta sẽ thu được *between-class variance* $(m_1 - m_2)^2$, lớn và *within-class variance*, $s_1^2 + s_2^2$ nhỏ.

Tiếp theo, chúng ta sẽ tìm biểu thức phụ thuộc giữa tử số và mẫu số trong vế phải của (22.5) vào \mathbf{w} . Với tử số,

$$(m_1 - m_2)^2 = (\mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2))^2 = \mathbf{w}^T \underbrace{(\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T}_{\mathbf{S}_B} \mathbf{w} = \mathbf{w}^T \mathbf{S}_B \mathbf{w} \quad (22.6)$$

\mathbf{S}_B còn được gọi là ma trận **between-class covariance**. Có thể nhận thấy đây là một ma trận đối xứng nửa xác định dương. Với mẫu số,

$$\begin{aligned} s_1^2 + s_2^2 &= \sum_{k=1}^2 \sum_{n \in \mathcal{C}_k} (\mathbf{w}^T(\mathbf{x}_n - \mathbf{m}_k))^2 \\ &= \mathbf{w}^T \underbrace{\sum_{k=1}^2 \sum_{n \in \mathcal{C}_k} (\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T}_{\mathbf{S}_W} \mathbf{w} = \mathbf{w}^T \mathbf{S}_W \mathbf{w} \end{aligned} \quad (22.7)$$

\mathbf{S}_W còn được gọi là ma trận **within-class covariance**. Đây cũng là một ma trận đối xứng nửa xác định dương vì nó là tổng của hai ma trận đối xứng nửa xác định dương².

Như vậy, bài toán tối ưu cho LDA trở thành

$$\mathbf{w} = \arg \max_{\mathbf{w}} \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} \quad (22.8)$$

22.2.3 Nghiệm của bài toán tối ưu

Nghiệm \mathbf{w} của (22.8) sẽ là nghiệm của phương trình đạo hàm hàm mục tiêu bằng 0. Sử dụng quy tắc chuỗi cho đạo hàm hàm nhiều biến và công thức $\nabla_{\mathbf{w}} \mathbf{w}^T \mathbf{A} \mathbf{w} = 2\mathbf{A} \mathbf{w}$ nếu \mathbf{A} là một ma trận đối xứng, ta thu được

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{(\mathbf{w}^T \mathbf{S}_W \mathbf{w})^2} (2\mathbf{S}_B \mathbf{w} (\mathbf{w}^T \mathbf{S}_W \mathbf{w}) - 2\mathbf{w}^T \mathbf{S}_B \mathbf{w}^T \mathbf{S}_W \mathbf{w}) = \mathbf{0} \quad (22.9)$$

$$\Leftrightarrow \mathbf{S}_B \mathbf{w} = \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} \mathbf{S}_W \mathbf{w} = J(\mathbf{w}) \mathbf{S}_W \mathbf{w} \quad (22.10)$$

$$\Rightarrow \mathbf{S}_W^{-1} \mathbf{S}_B \mathbf{w} = J(\mathbf{w}) \mathbf{w} \quad (22.11)$$

Lưu ý: Trong (22.11), ta đã giả sử rằng ma trận \mathbf{S}_W là khả nghịch. Điều này không luôn luôn đúng, nhưng có một kỹ thuật nhỏ là xấp xỉ \mathbf{S}_W bởi $\tilde{\mathbf{S}}_W = \mathbf{S}_W + \lambda \mathbf{I}$ với λ là một số thực dương nhỏ. Ma trận mới này là khả nghịch vì trị riêng nhỏ nhất của nó bằng với trị riêng nhỏ nhất của \mathbf{S}_W cộng với λ tức không nhỏ hơn $\lambda > 0$. Điều này được suy ra từ việc \mathbf{S}_W

² Trong (22.6) và (22.7), chúng ta đã sử dụng đẳng thức $(\mathbf{a}^T \mathbf{b})^2 = (\mathbf{a}^T \mathbf{b})(\mathbf{a}^T \mathbf{b}) = \mathbf{a}^T \mathbf{b} \mathbf{b}^T \mathbf{a}$ với \mathbf{a}, \mathbf{b} là hai vectors cùng chiều bất kỳ.

là một ma trận nửa xác định dương. Từ đó, $\bar{\mathbf{S}}_W$ là một ma trận xác định dương vì mọi trị riêng của nó là không nhỏ hơn λ , và vì thế, nó khả nghịch. Khi tính toán, ta có thể sử dụng nghịch đảo của $\bar{\mathbf{S}}_W$. Kỹ thuật này được sử dụng rất nhiều khi cần sử dụng nghịch đảo của một ma trận nửa xác định dương và chưa biết nó có thực sự là xác định dương hay không.

Quay trở lại với đẳng thức (22.11), vì $J(\mathbf{w})$ là một số vô hướng, ta suy ra \mathbf{w} phải là một vector riêng của $\mathbf{S}_W^{-1}\mathbf{S}_B$ ứng với $J(\mathbf{w})$. Vậy, để hàm mục tiêu là lớn nhất thì $J(\mathbf{w})$ chính là trị riêng lớn nhất của $\mathbf{S}_W^{-1}\mathbf{S}_B$. Dấu bằng xảy ra khi \mathbf{w} là vector riêng ứng với trị riêng lớn nhất đó.

Từ có thể thấy ngay rằng nếu \mathbf{w} là nghiệm của (22.8) thì $k\mathbf{w}$ cũng là nghiệm với k là số thực khác không bất kỳ. Vậy ta có thể chọn \mathbf{w} sao cho $(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w} = L$ với L là trị riêng lớn nhất của $\mathbf{S}_W^{-1}\mathbf{S}_B$ và cũng là giá trị tối ưu của $J(\mathbf{w})$. Khi đó, thay định nghĩa của \mathbf{S}_B ở (22.6) vào (22.11) ta có:

$$L\mathbf{w} = \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2) \underbrace{(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w}}_L = L\mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2) \quad (22.12)$$

Điều này nghĩa là ta có thể chọn

$$\mathbf{w} = \alpha \mathbf{S}_W^{-1}(\mathbf{m}_1 - \mathbf{m}_2) \quad (22.13)$$

với $\alpha \neq 0$ bất kỳ. Biểu thức (22.13) còn được biết như là *Fisher's linear discriminant*, được đặt theo tên nhà khoa học Ronald Fisher (<https://goo.gl/eUk1KS>).

22.3 LDA cho bài toán phân lớp nhiều lớp

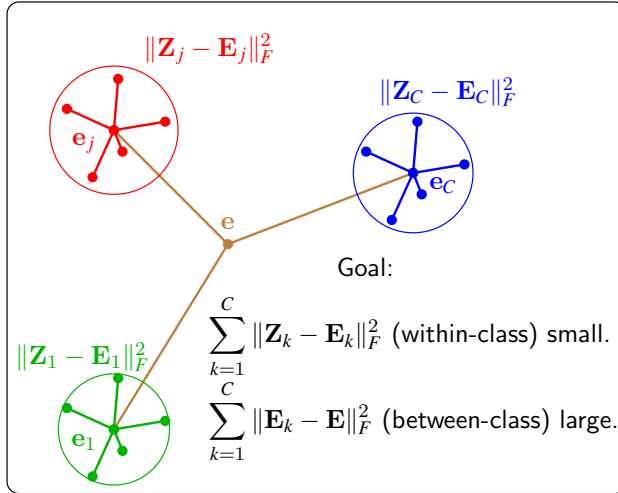
22.3.1 Xây dựng hàm mục tiêu

Trong mục này, chúng ta sẽ xem xét trường hợp tổng quát của LDA, được gọi là *multi-class LDA*, khi có nhiều hơn hai lớp dữ liệu, $C > 2$. Giả sử rằng chiều của dữ liệu D lớn hơn C . Giả sử rằng chiều mà chúng ta muốn giảm về là $D' < D$ và dữ liệu mới ứng với mỗi điểm dữ liệu \mathbf{x} là:

$$\mathbf{z} = \mathbf{W}^T \mathbf{x} \quad (22.14)$$

với $\mathbf{W} \in \mathbb{R}^{D \times D'}$. Một vài ký hiệu:

- $\mathbf{X}_k, \mathbf{Z}_k = \mathbf{W}^T \mathbf{X}_k$ lần lượt là ma trận dữ liệu của lớp thứ k trong không gian ban đầu và không gian mới với số chiều nhỏ hơn. Mỗi cột tương ứng với một điểm dữ liệu.
- $\mathbf{m}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{x}_k \in \mathbb{R}^D$ là vector kỳ vọng của lớp thứ k trong không gian ban đầu.
- $\mathbf{e}_k = \frac{1}{N_k} \sum_{n \in \mathcal{C}_k} \mathbf{z}_n = \mathbf{W}^T \mathbf{m}_k \in \mathbb{R}^{D'}$ là vector kỳ vọng của lớp thứ k trong không gian mới.
- $\mathbf{m} \in \mathbb{R}^D$ là vector trung bình của toàn bộ dữ liệu trong không gian ban đầu và $\mathbf{e} \in \mathbb{R}^{D'}$ là vector trung bình trong không gian mới.



Hình 22.3: LDA cho multi-class classification problem. Mục đích cũng là sự khác nhau giữa các thành phần trong một lớp (within-class) là nhỏ và sự khác nhau giữa các lớp là lớn. Các điểm dữ liệu có màu khác nhau thể hiện các lớp khác nhau.

Một trong những cách xây dựng hàm mục tiêu cho multi-class LDA được minh họa trong Hình 22.3. Độ phân tán của một tập hợp dữ liệu có thể được coi như tổng bình phương khoảng cách từ mỗi điểm tới vector kỳ vọng của chúng. Nếu tất cả các điểm đều gần vector trung bình của chúng thì độ phân tán của tập dữ liệu đó được coi là nhỏ. Ngược lại, nếu tổng này là lớn, tức trung bình các điểm đều xa trung tâm, tập hợp này có thể được coi là có độ phân tán cao. Dựa vào nhận xét này, ta có thể xây dựng các đại lượng within- và between-class variance như dưới đây.

Within-class variance của lớp thứ k có thể được tính như sau:

$$\sigma_k^2 = \sum_{n \in \mathcal{C}_k} \|\mathbf{z}_n - \mathbf{e}_k\|_F^2 = \|\mathbf{Z}_k - \mathbf{E}_k\|_F^2 = \|\mathbf{W}^T(\mathbf{X}_k - \mathbf{M}_k)\|_F^2 \quad (22.15)$$

$$= \text{trace}(\mathbf{W}^T(\mathbf{X}_k - \mathbf{M}_k)(\mathbf{X}_k - \mathbf{M}_k)^T \mathbf{W}) \quad (22.16)$$

Với \mathbf{E}_k một ma trận có các cột giống hệt nhau và bằng với vector trung bình \mathbf{e}_k . Có thể nhận thấy $\mathbf{E}_k = \mathbf{W}^T \mathbf{M}_k$ với \mathbf{M}_k là ma trận có các cột giống hệt nhau và bằng với vector trung bình \mathbf{m}_k trong không gian ban đầu. Vậy đại lượng đo within-class trong multi-class LDA có thể được đo bằng:

$$s_W = \sum_{k=1}^C \sigma_k^2 = \sum_{k=1}^C \text{trace}(\mathbf{W}^T(\mathbf{X}_k - \mathbf{M}_k)(\mathbf{X}_k - \mathbf{M}_k)^T \mathbf{W}) = \text{trace}(\mathbf{W}^T \mathbf{S}_W \mathbf{W}) \quad (22.17)$$

với

$$\mathbf{S}_W = \sum_{k=1}^C \|\mathbf{X}_k - \mathbf{M}_k\|_F^2 = \sum_{k=1}^C \sum_{n \in \mathcal{C}_k} (\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^T \quad (22.18)$$

Ma trận \mathbf{S}_W này là một ma trận nửa xác định dương.

Between-class variance lớn có thể đạt được nếu tất cả các điểm trong không gian mới đều xa vector trung bình chung \mathbf{e} . Việc này cũng có thể đạt được nếu các vector trung bình của mỗi lớp xa các vector trung bình chung (trong không gian mới). Vậy ta có thể định nghĩa đại lượng between-class như sau:

$$s_B = \sum_{k=1}^C N_k \|\mathbf{e}_k - \mathbf{e}\|_F^2 = \sum_{k=1}^C \|\mathbf{E}_k - \mathbf{E}\|_F^2 \quad (22.19)$$

Ta lấy N_k làm trọng số vì có thể có những lớp có nhiều phần tử so với các lớp còn lại. Chú ý rằng ma trận \mathbf{E} có thể có số cột *linh động*, phụ thuộc vào số cột của ma trận \mathbf{E}_k mà nó đi cùng (và bằng N_k).

Lập luận tương tự như (22.17), bạn đọc có thể chứng minh được:

$$s_B = \text{trace}(\mathbf{W}^T \mathbf{S}_B \mathbf{W}) \quad (22.20)$$

với

$$\mathbf{S}_B = \sum_{k=1}^C (\mathbf{M}_k - \mathbf{M})(\mathbf{M}_k - \mathbf{M})^T = \sum_{k=1}^C N_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T \quad (22.21)$$

và số cột của ma trận \mathbf{M} cũng *linh động* theo số cột của \mathbf{M}_k . Ma trận này là tổng của các ma trận đối xứng nửa xác định dương, nên nó là một ma trận đối xứng nửa xác định dương.

22.3.2 Hàm mục tiêu cho multi-class LDA

Với cách định nghĩa và ý tưởng về within-class variance nhỏ và between-class variance lớn như trên, ta có thể xây dựng bài toán tối ưu

$$\mathbf{W} = \arg \max_{\mathbf{W}} J(\mathbf{W}) = \arg \max_{\mathbf{W}} \frac{\text{trace}(\mathbf{W}^T \mathbf{S}_B \mathbf{W})}{\text{trace}(\mathbf{W}^T \mathbf{S}_W \mathbf{W})} \quad (22.22)$$

Nghiệm cũng được tìm bằng cách giải phương trình đạo hàm mục tiêu bằng 0. Nhắc lại về đạo hàm của hàm trace theo ma trận:

$$\nabla_{\mathbf{W}} \text{trace}(\mathbf{W}^T \mathbf{A} \mathbf{W}) = 2\mathbf{A} \mathbf{W} \quad (22.23)$$

với $\mathbf{A} \in \mathbb{R}^{D \times D}$ là một ma trận đối xứng.

Với cách tính tương tự như (22.9) - (22.11), ta có:

$$\nabla_{\mathbf{W}} J(\mathbf{W}) = \frac{2(\mathbf{S}_B \mathbf{W} \text{trace}(\mathbf{W}^T \mathbf{S}_W \mathbf{W}) - \text{trace}(\mathbf{W}^T \mathbf{S}_B \mathbf{W}) \mathbf{S}_W \mathbf{W})}{(\text{trace}(\mathbf{W}^T \mathbf{S}_W \mathbf{W}))^2} = \mathbf{0} \quad (22.24)$$

$$\Leftrightarrow \mathbf{S}_W^{-1} \mathbf{B} \mathbf{W} = J(\mathbf{W}) \mathbf{W} \quad (22.25)$$

Từ đó suy ra mỗi cột của \mathbf{W} là một vector riêng của $\mathbf{S}_W^{-1} \mathbf{S}_B$ ứng với trị riêng lớn nhất của ma trận này. Nhận thấy rằng các cột của \mathbf{W} cần phải độc lập tuyến tính. Vì nếu không, dữ liệu trong không gian mới $\mathbf{z} = \mathbf{W}^T \mathbf{x}$ sẽ phụ thuộc tuyến tính và có thể tiếp tục được giảm số chiều. Vậy các cột của \mathbf{W} là các vector độc lập tuyến tính ứng với trị riêng cao nhất của $\mathbf{S}_W^{-1} \mathbf{S}_B$. Câu hỏi đặt ra là: Có nhiều nhất bao nhiêu vector riêng độc lập tuyến tính ứng với trị riêng lớn nhất của $\mathbf{S}_W^{-1} \mathbf{S}_B$? Số lượng này chính là số chiều D' của dữ liệu mới.

Số lượng lớn nhất các vector riêng độc lập tuyến tính ứng với một trị riêng của một ma trận không thể lớn hơn rank của ma trận đó. Dưới đây là một bổ đề quan trọng.

Bổ đề:

$$\text{rank}(\mathbf{S}_B) \leq C - 1 \tag{22.26}$$

Chứng minh³:

Viết lại 22.21 dưới dạng

$$\mathbf{S}_B = \mathbf{P}\mathbf{P}^T \tag{22.27}$$

với $\mathbf{P} \in R^{D \times C}$ mà cột thứ k của nó là $\mathbf{p}_k = \sqrt{N_k}(\mathbf{m}_k - \mathbf{m})$.

Hơn nữa, cột cuối cùng là một tổ hợp tuyến tính của các cột còn lại:

$$\mathbf{m}_C - \mathbf{m} = \mathbf{m}_C - \frac{\sum_{k=1}^C N_k \mathbf{m}_k}{N} = \sum_{k=1}^{C-1} \frac{N_k}{N} (\mathbf{m}_k - \mathbf{m}) \tag{22.28}$$

Như vậy ma trận \mathbf{P} có nhiều nhất $C - 1$ cột độc lập tuyến tính, vậy nên rank⁴ của nó không vượt quá $C - 1$. Cuối cùng, \mathbf{S}_B là tích của hai ma trận với rank không quá $C - 1$, nên $\text{rank}(\mathbf{S}_B)$ không vượt quá $C - 1$. □

Từ đó ra có $\text{rank}(\mathbf{S}_W^{-1}\mathbf{S}_B) \leq \text{rank}\mathbf{S}_B \leq C - 1$. Vậy số chiều của không gian mới là một số không lớn hơn $C - 1$.

Tóm lại, nghiệm của bài toán multi-class LDA là các vector riêng độc lập tuyến tính ứng với trị riêng cao nhất của $\mathbf{S}_W^{-1}\mathbf{S}_B$.

Lưu ý: Có nhiều cách khác nhau để xây dựng hàm mục tiêu cho multi-class LDA dựa trên việc định nghĩa within-class variance nhỏ và between-class variance lớn. Chúng ta đang sử dụng hàm trace để đo lường hai đại lượng này. Một ví dụ khác về hàm tối ưu là $J(\mathbf{W}) = \text{trace}(s_W^{-1} s_B) = \text{trace}\{(\mathbf{W}\mathbf{S}_W\mathbf{W}^T)^{-1}(\mathbf{W}\mathbf{S}_B\mathbf{W}^T)\}$ [Fuk13]. Hàm số này cũng đạt giá trị lớn nhất khi \mathbf{W} là tập hợp của D' vector riêng ứng với các trị riêng lớn nhất của $\mathbf{S}_W^{-1}\mathbf{S}_B$. Có một điểm chung giữa các cách tiếp cận này là chiều của không gian mới sẽ không vượt quá $C - 1$.

22.4 Ví dụ trên Python

Trong mục này, chúng ta sẽ minh họa LDA cho bài toán phân lớp nhị phân qua một ví dụ đơn giản với dữ liệu trong không gian hai chiều.

Dữ liệu của hai lớp được tạo như sau:

³ Việc chứng minh này không thực sự quan trọng, chỉ phù hợp với những bạn muốn hiểu sâu.

⁴ Các tính chất của rank có thể được tìm thấy trong Mục 1.8.

```

from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages
np.random.seed(22)

means = [[0, 5], [5, 0]]
cov0 = [[4, 3], [3, 4]]
cov1 = [[3, 1], [1, 1]]
N0, N1 = 50, 40
N = N0 + N1
X0 = np.random.multivariate_normal(means[0], cov0, N0) # each row is a data point
X1 = np.random.multivariate_normal(means[1], cov1, N1)

```

Các điểm dữ liệu của hai lớp được minh họa bởi các màu khác nhau trên Hình 22.4.

Tiếp theo, chúng ta đi tính các ma trận within-class và between-class covariance:

```

# Build S_B
m0 = np.mean(X0.T, axis = 1, keepdims = True)
m1 = np.mean(X1.T, axis = 1, keepdims = True)

a = (m0 - m1)
S_B = a.dot(a.T)

# Build S_W
A = X0.T - np.tile(m0, (1, N0))
B = X1.T - np.tile(m1, (1, N1))

S_W = A.dot(A.T) + B.dot(B.T)

```

Nghiệm của bài toán là vector riêng ứng với trị riêng lớn nhất của $\mathbf{S}_W^{-1}\mathbf{W}_B$:

```

_, W = np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
w = W[:,0]
print('w = ', w)

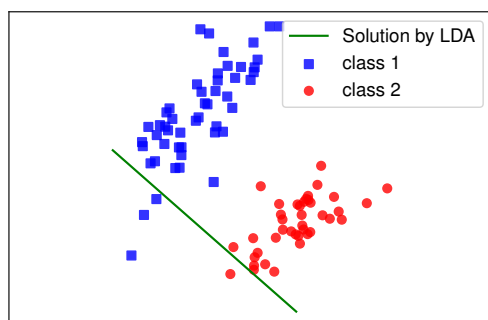
```

Kết quả:

```
w = [ 0.75091074 -0.66040371]
```

Đường thẳng có phương w được minh họa bởi đường màu lục trên Hình 22.4. Ta thấy rằng nghiệm này hợp lý với dữ liệu của bài toán.

Để kiểm chứng độ chính xác của nghiệm tìm được, ta cùng so sánh nó với nghiệm tìm được bởi thư viện **sklearn**.



Hình 22.4: Ví dụ minh họa về LDA trong không gian hai chiều. Đường thẳng màu lục là đường thẳng mà dữ liệu sẽ được chiếu lên. Ta có thể thấy rằng, nếu chiếu lên đường thẳng này, dữ liệu của hai lớp sẽ nằm về hai phía của một điểm trên đường thẳng đó.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
X = np.concatenate((X0, X1))
y = np.array([0]*N0 + [1]*N1)
clf = LinearDiscriminantAnalysis()
clf.fit(X, y)

print('w_sklearn = ', clf.coef_[0]/np.linalg.norm(clf.coef_)) # normalize
```

```
w_sklearn = [ 0.75091074 -0.66040371]
```

Ta thấy rằng nghiệm tìm theo công thức và nghiệm tìm theo thư viện là như nhau.

Một ví dụ khác so sánh PCA và LDA có thể được tìm thấy tại *Comparison of LDA and PCA 2D projection of Iris dataset* (<https://goo.gl/tWjAEs>).

22.5 Thảo luận

- LDA là một phương pháp giảm chiều dữ liệu có sử dụng thông tin về label của dữ liệu. Vì vậy, LDA là một thuật toán supervised.
- Ý tưởng cơ bản của LDA là tìm một không gian mới với số chiều nhỏ hơn không gian ban đầu sao cho hình chiếu của các điểm trong cùng lớp lên không gian mới này là gần nhau trong khi hình chiếu của các điểm của các lớp khác nhau là khác nhau.
- Trong PCA, số chiều của không gian mới có thể là bất kỳ số nào không lớn hơn số chiều và số điểm của dữ liệu. Trong LDA, với bài toán có C classes, số chiều của không gian mới chỉ có thể không vượt quá $C - 1$.
- Với bài toán có hai lớp, từ Hình 22.1 ta có thể thấy rằng hai lớp là linearly separable nếu và chỉ nếu tồn tại một đường thẳng và một điểm trên đường thẳng đó (điểm mùa lục) sao cho dữ liệu hình chiếu trên đường thẳng của hai lớp nằm về hai phía khác nhau của điểm đó.

- LDA hoạt động rất tốt nếu các lớp là linearly separable. Chất lượng mô hình giảm đi rõ rệt nếu các classes là không linearly separable. Điều này dễ hiểu vì khi đó, chiếu dữ liệu lên phương nào thì cũng bị chồng lấn, và việc tách biệt không thể thực hiện được như ở không gian ban đầu.
- Mặc dù có hạn chế, ý tưởng về *small within-class variance* và *big within-class variance* còn được gọi là *Fisher's optimization criterion*, được sử dụng rất nhiều trong các thuật toán phân lớp [VM17, VM16, YZFZ11].

Convex optimization—Tối ưu lồi

Tập lồi và hàm lồi

23.1 Giới thiệu

Các bài toán tối ưu đã thảo luận trong cuốn sách này đều là các *bài toán tối ưu không ràng buộc* (*unconstrained optimization problems*), tức tối ưu hàm mất mát mà không có *điều kiện ràng buộc* (*constraints*) nào về nghiệm. Tuy nhiên, không chỉ trong Machine Learning, trên thực tế các bài toán tối ưu thường có rất nhiều ràng buộc khác nhau.

Trong toán 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 một *feasible point*, các điểm không trong *feasible set* được gọi là các *infeasible point*.

Chú ý:

- Nếu bài toán là tìm giá trị lớn nhất thay vì nhỏ nhất của hàm mục tiêu, 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*.



Example of convex sets

Hình 23.1: Các ví dụ về tập lồi.

- 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 chương này, \mathbf{x}, \mathbf{y} được dùng chủ yếu để ký hiệu các biến số, không phải là dữ liệu như trong các chương 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 rất nhiều bài chưa có lời giải hiểu quả. Hầu hết các phương pháp tìm nghiệm không chứng minh được nghiệm tìm được có phải là tức đúng là điểm làm cho hàm số đạt giá trị nhỏ nhất hay lớn nhất hay không (*global optimal*). Thay vào đó, nghiệm thường là các *điểm cực trị* (*local optimal*). Trong nhiều trường hợp, các nghiệm *local optimal* cũng mang lại những kết quả tốt.

Để bắt đầu nghiên cứu về tối ưu, chúng ta cần biết tới 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 chương này, chúng ta sẽ thảo luận định nghĩa và các tính chất cơ bản của *tập lồi* và *hàm lồi*. *Bài toán tối ưu lồi* (*convex optimization problems*) sẽ được đề cập trong chương tiếp theo.

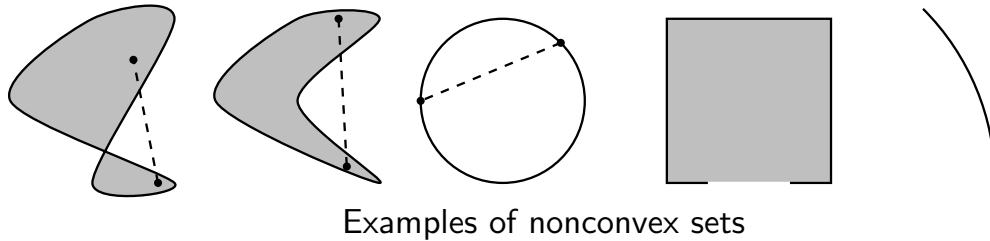
23.2 Tập lồi – Convex sets

23.2.1 Định nghĩa

Bạn đọc có thể đã biết đến khái niệm *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 không chính thức của tập lồi: Một tập hợp được gọi là *tập lồi* nếu mọi điểm trên đoạn thẳng nối hai điểm *bất kỳ* trong tập hợp đó đều thuộc tập hợp đó.

Một vài ví dụ về tập lồi được cho trong Hình 23.1. Các hình với đường biên màu đen thể hiện việc biên cũng thuộc vào hình đó, biên màu trắng thể hiện việc biên đó không nằm trong hình. Đường thẳng hoặc đoạn thẳng cũng là một tập lồi theo định nghĩa phía trên.



Hình 23.2: Các ví dụ về tập không lồi.

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 trên trần nhà, mọi điểm trong căn phòng đều được chiếu sáng.
- Nếu một đất nước có bản đồ dạng một hình lồi thì đoạn thẳng nối hai thành phố bất kỳ trong đất nước đó nằm trọn vẹn trong nước đó. Một cách lý tưởng, mọi đường bay trong đất nước đều được tối ưu vì chi phí bay thẳng ít hơn chi phí bay đường vòng hoặc qua không phận của nước khác. Bản đồ Việt Nam không có dạng lồi vì đường thẳng nối sân bay Nội Bài và Tân Sơn Nhất đi qua địa phận Campuchia.

Hình 23.2 minh họa một vài ví dụ về các tập không phải là tập lồi, nói gọn là *tập không lồi* (*nonconvex set*). 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 một *tập lồi* vì đoạn thẳng nối hai điểm ở đáy 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 23.1: Convex set–Tập hợp lồi

Một tập hợp \mathcal{C} được gọi là một *tập lồi* 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 thẳng nào cũng nằm trong không gian đó. Tập rỗng cũng có thể coi là một trường hợp đặc biệt của *tập lồi*.

23.2.2 Các ví dụ về tập lỗi

Siêu mặt phẳng và nửa không gian

Định nghĩa 23.2: Hyperplane–Siêu mặt phẳng

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

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{a}^T \mathbf{x} = b \quad (23.1)$$

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

Hyperplanes là các *tập lỗi*. Điều này có thể được suy ra từ Định nghĩa 23.1. Thật vậy, nếu

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

thì với $0 \leq \theta \leq 1$ bất kỳ, ta có $\mathbf{a}^T \mathbf{x}_\theta = \mathbf{a}^T(\theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2) = \theta b + (1 - \theta)b = b$

Định nghĩa 23.3: Halfspace–Nửa không gian

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

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{a}^T \mathbf{x} \leq b$$

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

Các halfspace cũng là các tập lỗi, bạn đọc có thể dễ dàng nhận thấy theo Định nghĩa 23.1 và cách chứng minh tương tự như trên.

Norm balls

Định nghĩa 23.4: Norm ball

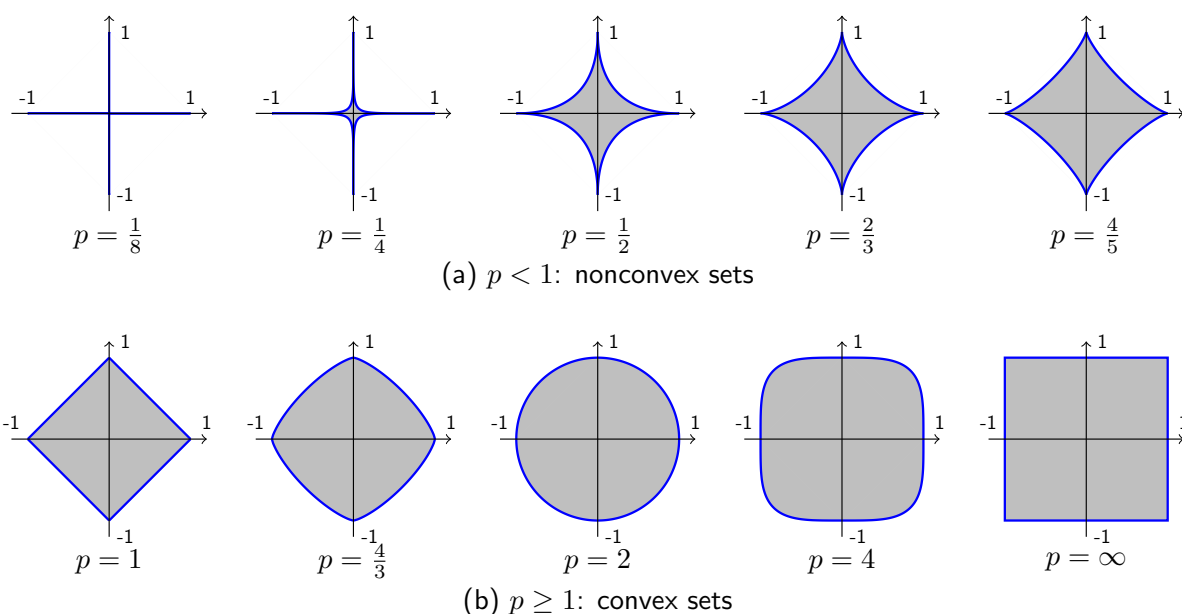
Cho một tâm \mathbf{x}_c và một bán kính r và khoảng cách giữa các điểm được xác định bởi một norm. *Norm ball* tương ứng là tập hợp các điểm thỏa mãn

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

Khi norm là ℓ_2 norm, ta có norm ball là một hình tròn trong không gian hai chiều, hình cầu trong không gian ba chiều, hoặc siêu cầu trong các không gian nhiều chiều. Khi dùng ℓ_2 norm, norm ball được gọi là *Euclidean norm*.

Norm balls là các tập lỗi. Để chứng minh việc này, ta dùng Định nghĩa 23.1 và bất đẳng thức tam giác 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, xét

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



Hình 23.3: Hình dạng của các tập hợp bị chặn bởi (a) các pseudo-norm và (b) các norm.

$\mathbf{x}_\theta = \theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2$, ta có

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

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

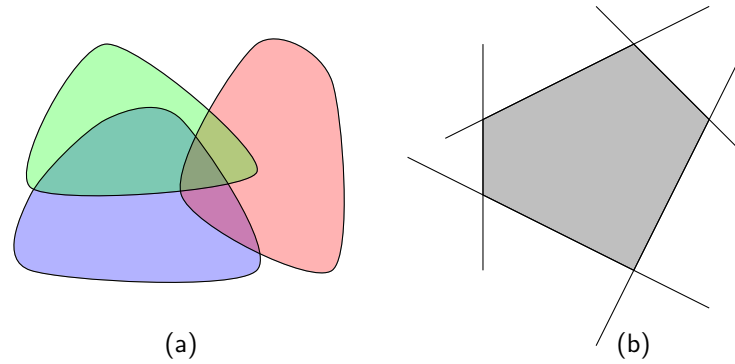
Hình 23.3 minh họa tập hợp các điểm có tọa độ (x, y) trong không gian hai chiều thỏa mãn:

$$(|x|^p + |y|^p)^{1/p} \leq 1 \tag{23.2}$$

với hàng trên là các tập với $0 < p < 1$, là các pseudo-norm, và hàng dưới tương ứng với $p \geq 1$, là các norm thực sự. Chúng ta có thể thấy rằng khi p nhỏ gần bằng 0, tập hợp các điểm thỏa mãn bất đẳng thức (23.2) 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 pseudo-norm 0. 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 ℓ_p norm.

23.2.3 Giao của các tập lỗi

Giao của các tập lỗi là một tập lỗi. Việc này có thể nhận ra trong Hình 23.4a. Giao của hai trong ba hoặc cả ba tập lỗi đều là các tập lỗi. Việc này có thể được chứng minh theo Định nghĩa 23.1. Nếu $\mathbf{x}_1, \mathbf{x}_2$ thuộc vào giao của các tập lỗi, tức thuộc tất cả các tập lỗi đã cho, thì $(\theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2)$ cũng thuộc vào tất cả các tập lỗi, tức thuộc vào giao của chúng.



Hình 23.4: (a) Giao của các tập lồi là một tập lồi. (b) Giao của các hyperplanes và halfspace là một tập lồi và được gọi là polyhedron (số nhiều là polyhedra).

Từ đó suy ra giao của các *halfspaces* và các *hyperplanes* cũng là một tập lồi. Chúng là các đa giác lồi trong không gian hai chiều và đa diện lồi trong không gian ba chiều. Trong không gian nhiều chiều, giao của các *halfspaces* và *hyperplanes* được gọi là **polyhedra**. Giả sử có m *halfspace* và p *hyperplanes*. Mỗi một *halfspace* có thể được viết dưới dạng $\mathbf{a}_i^T \mathbf{x} \leq b_i, \forall i = 1, 2, \dots, m$. Mỗi một *hyperplane* có thể được 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.

23.2.4 Tổ hợp lồi và bao lồi

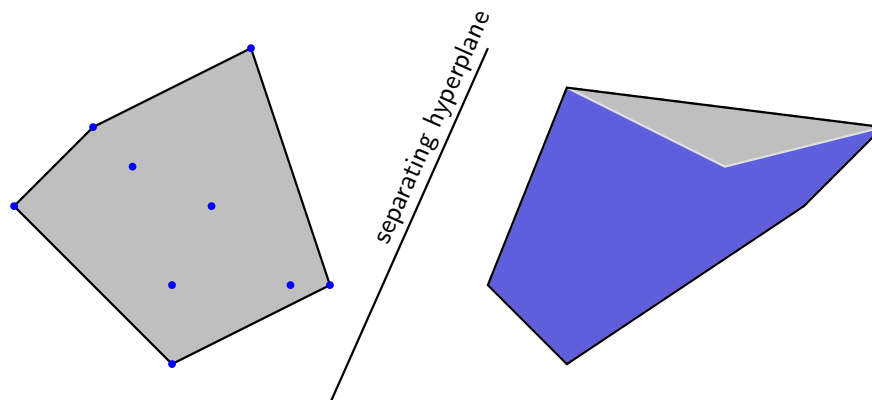
Định nghĩa 23.5: Tổ hợp lồi–Convex combination

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

$$\mathbf{x} = \theta_1 \mathbf{x}_1 + \theta_2 \mathbf{x}_2 + \dots + \theta_k \mathbf{x}_k, \quad \text{với } \theta_1 + \theta_2 + \dots + \theta_k = 1 \text{ và } \theta_i \geq 0, \forall i = 1, 2, \dots, k$$

Bao lồi (*convex hull*) của một **tập hợp bất kỳ** là tập hợp tất cả các điểm là *convex combination* của tập hợp đó. *Convex hull* của một tập bất kỳ là một *convex set*. *Convex hull* của một *convex set* là chính nó. *Convex hull* của một tập hợp chính là *convex set* **nhỏ nhất** chứa tập hợp đó. Khái niệm **nhỏ nhất** được hiểu là mọi tập lồi chứa toàn bộ một tập hợp bất kỳ đều chứa *convex hull* của tập hợp đó.

Nhắc lại về khái niệm *linear separable* đã sử dụng nhiều trong cuốn sách. Hai tập hợp được gọi là *linearly separable* nếu các *convex hull* của chúng không có điểm chung.



Hình 23.5: Trái: Giao của các tập lồi là một tập lồi. Phải: giao của các hyperplanes và halfspaces là một tập lồi và được gọi là polyhedron (số nhiều là polyhedra).

Trong Hình 23.5, convex hull của các điểm màu xanh là vùng màu xám bao bởi các đa giác lồi. Ở Hình 23.5 bên phải phải, convex hull của đa giác màu xanh là hợp của nó và phần tam giác màu xám.

Định lý 23.1: Siêu phẳng phân chia—Separating hyperplane theorem

Hai tập lồi không rỗng \mathcal{C}, \mathcal{D} là không giao nhau nếu và chỉ nếu tồn tại một vector \mathbf{a} và một số b sao cho

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

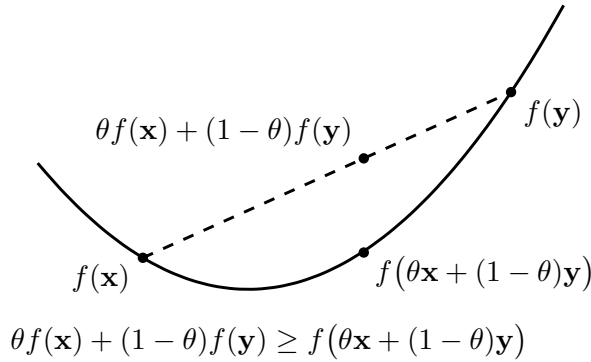
Tập hợp tất cả các điểm \mathbf{x} thỏa mãn $\mathbf{a}^T \mathbf{x} = b$ chính là một hyperplane. Hyperplane này được gọi là *separating hyperplane*.

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, bạn đọc được khuyến khích đọc thêm Chương 2 của cuốn Convex Optimization [BV04].

23.3 Convex functions

23.3.1 Định nghĩa

Trước hết ta xem xét các hàm một biến với 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 23.6). *Tập xác định (domain)* của một hàm số $f(\cdot)$ thường được ký hiệu là $\text{dom}f$.



Hình 23.6: 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.

Định nghĩa 23.6: Convex function–Hàm lồi

Một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ được gọi là một *hàm lồi* 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 $\mathbf{x}, \mathbf{y} \in \text{dom} f, 0 \leq \theta \leq 1$.

Điều kiện $\text{dom} f$ là một *tập lồi* là rất quan trọng. Nếu không có điều kiện này, tồn tại những θ mà $\theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2$ không thuộc $\text{dom} f$, và sẽ không định nghĩa được $f(\theta \mathbf{x} + (1 - \theta) \mathbf{y})$.

Một hàm số f được gọi là **concave** (tạm dịch là *lõm*) nếu $-f$ là *convex*. Một hàm số có thể không thuộc hai loại trên. Các hàm tuyến tính vừa *convex*, vừa *concave*.

Định nghĩa 23.7: Strictly convex function–Hàm lồi chặt

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

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

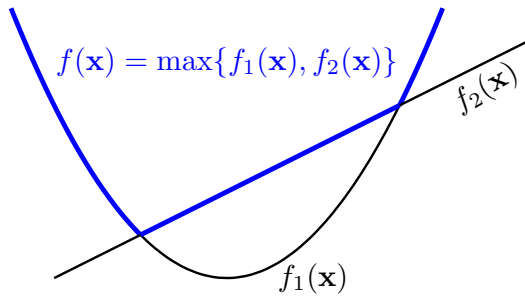
với mọi $\mathbf{x}, \mathbf{y} \in \text{dom} f, \mathbf{x} \neq \mathbf{y}, 0 < \theta < 1$ (chỉ khác với hàm *convex* ở dấu nhỏ hơn).

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

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*.

23.3.2 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 của hai hàm đã cho (nhắc lại rằng giao của hai tập lồi là một tập lồi)



Hình 23.7: Ví dụ về Pointwise maximum. Maximum của các hàm lồi là một hàm lồi.

- **Pointwise maximum và 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 *supremum*¹. Tính chất này có thể được chứng minh theo Định nghĩa 23.6. Hình 23.7 minh họa tính chất này. Các hàm $f_1(\mathbf{x}), f_2(\mathbf{x})$ là các hàm lồi. Đường màu xanh chính là đồ thị của hàm số $f(\mathbf{x}) = \max(f_1(\mathbf{x}), f_2(\mathbf{x}))$. Mọi đoạn thẳng nối hai điểm bất kỳ trên đường màu xanh đều *không nằm dưới* nó.

23.3.3 Ví dụ

Các hàm một biến

Ví dụ về các *convex functions* một biến:

- Hàm $y = ax + b$ là một *hàm lồi* vì đoạn thẳng nối hai điểm bất kỳ trên đường thẳng đó đều *không nằm phía dưới* đường thẳng 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.

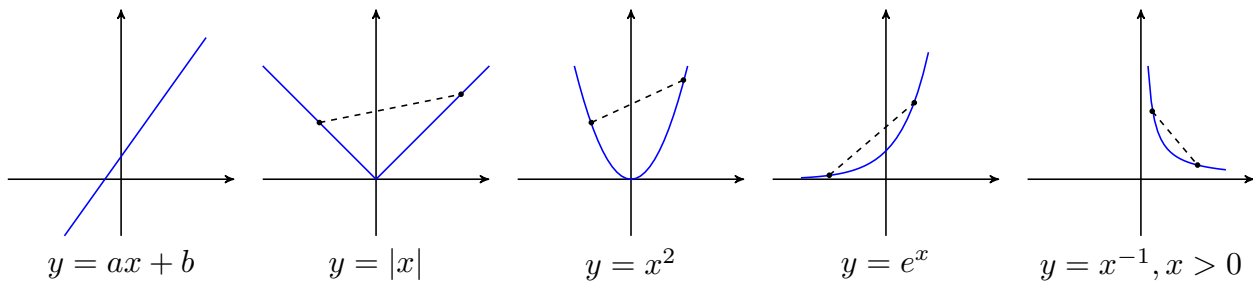
Hình 23.8 minh họa đồ thị của một số hàm *convex* thường gặp với biến một chiều.

Ví dụ về các *concave functions* một biến:

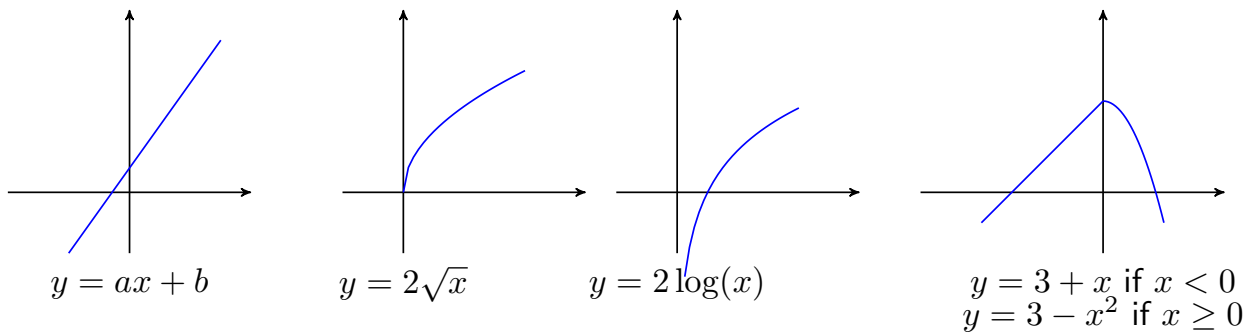
- Hàm $y = ax + b$ là một *concave function* vì $-y$ là một *convex function*.
- Hàm $y = x^a$ trên tập số dương và $0 \leq a \leq 1$.
- Hàm logarithm $y = \log(x)$ trên tập các số dương.

Hình 23.9 minh họa đồ thị của một vài hàm số *concave*.

¹ Xem *Infimum and Supremum* – Wikipedia (<https://goo.gl/AsX4oM>)



Hình 23.8: Ví dụ về các hàm convex một biến.



Hình 23.9: Ví dụ về các hàm concave một biến.

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, \mathbf{A} là một ma trận có cùng kích thước như \mathbf{X} để đảm bảo phép nhân ma trận thực hiện được và kết quả là một ma trận vuông.

Dạng toàn phương – Quadratic form

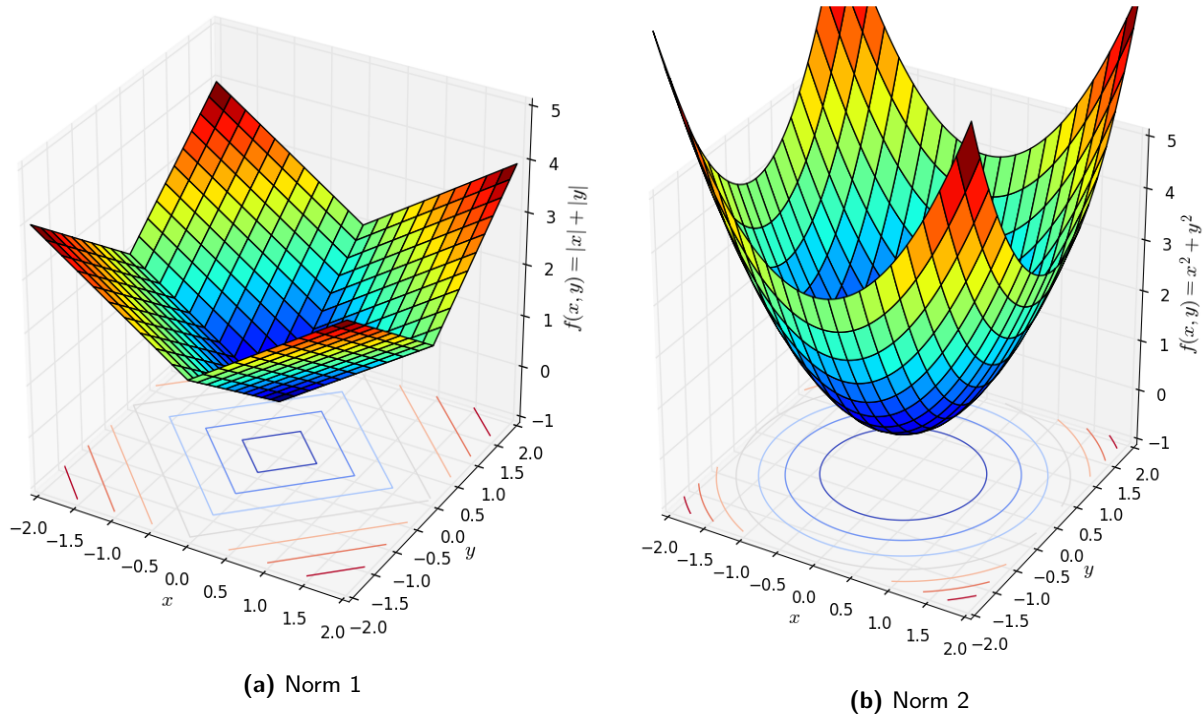
Hàm bậc hai một biến có dạng $f(x) = ax^2 + bx + c$ là convex nếu $a > 0$, là concave nếu $a < 0$.

Với biến là một vector $\mathbf{x} = [x_1, x_2, \dots, x_n]$, một dạng toàn phương (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} , \mathbf{b} là các ma trận và vector với chiều phù hợp và \mathbf{A} thường là một ma trận đối xứng.

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



Hình 23.10: Ví dụ về mặt của các norm hai biến.

Nhắc lại hàm mất mát trong linear regression có dạng

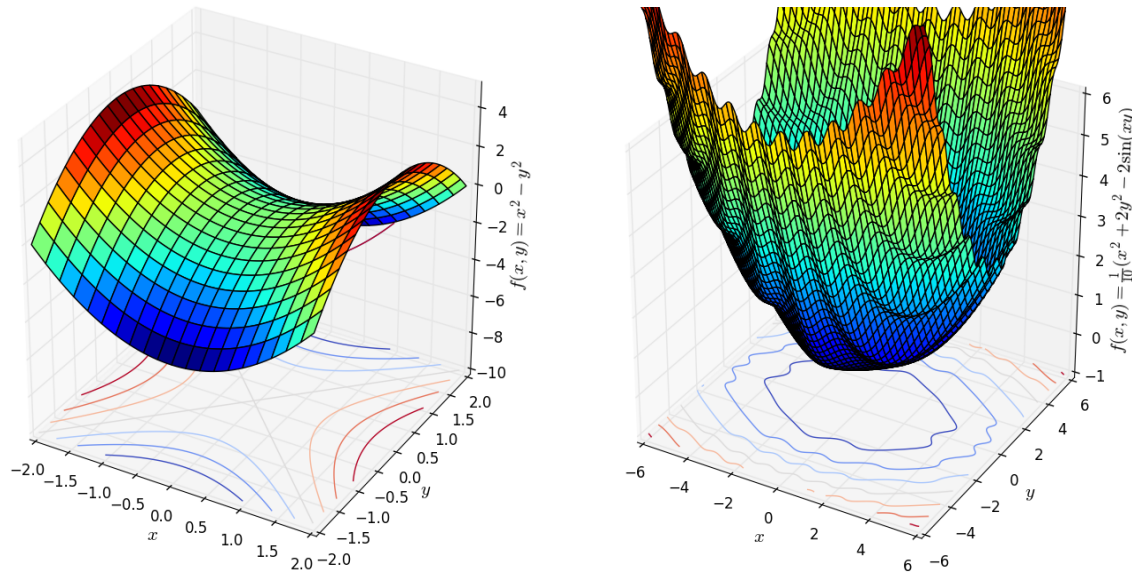
$$\begin{aligned} \mathcal{L}(\mathbf{w}) &= \frac{1}{2N} \|\mathbf{y} - \mathbf{X}^T \mathbf{w}\|_2^2 = \frac{1}{2N} (\mathbf{y} - \mathbf{X}^T \mathbf{w})^T (\mathbf{y} - \mathbf{X}^T \mathbf{w}) \\ &= \frac{1}{2N} \mathbf{w}^T \mathbf{X} \mathbf{X}^T \mathbf{w} - \frac{1}{N} \mathbf{y}^T \mathbf{X}^T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

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

Norms

Mọi hàm số bất kỳ thỏa mãn ba điều kiện của norm đều là convex. Việc này có thể được trực tiếp suy ra từ bất đẳng thức tam giác của một norm.

Hình 23.10 minh họa hai ví dụ về bề mặt của l_1 norm và l_2 norm trong không gian hai chiều (chiều thứ ba 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). 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 với learning rate phù hợp. Tính chất này khiến cho các hàm convex và strictly convex được đặc biệt quan tâm trong tối ưu.



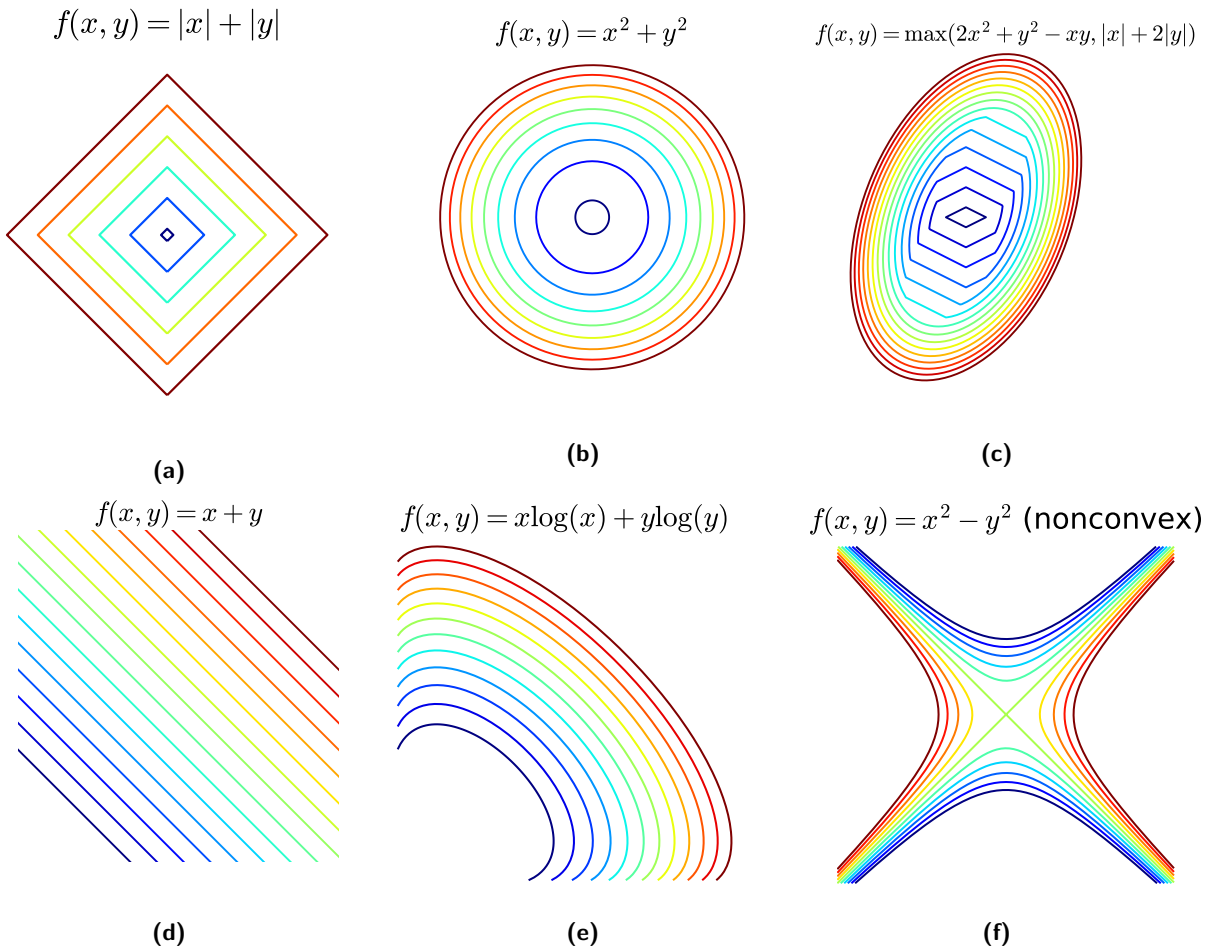
Hình 23.11: Ví dụ về các hàm hai biến không convex.

Hai hàm tiếp theo là ví dụ về các hàm không phải convex hay concave. Hàm thứ nhất $f(x, y) = x^2 - y^2$ là một hyperbolic, hàm thứ hai $f(x, y) = \frac{1}{10}(x^2 + 2y^2 - 2\sin(xy))$. Các bề mặt của hai hàm này được minh họa trên Hình 23.11

23.3.4 Contours–level sets

Để khảo sát tính lồi của các bề mặt trong không gian ba chiều, việc minh họa trực tiếp như các ví dụ trên đây có thể khó tưởng tượng hơn. Một phương pháp thường được sử dụng là dùng các *đường đồng mức* (*contour* hay *level set*). Contours là cách mô tả các mặt trong không gian ba chiều trong không gian hai chiều. Ở đó, các điểm thuộc cùng một *đường* tương ứng với các điểm làm cho hàm số có giá trị như nhau. Mỗi *đường* đó còn được gọi là một *level set*. Trong Hình 23.10 và Hình 23.11, các contour của các mặt trên mặt phẳng $0xy$ chính là các *level set*. Nói cách khác, mỗi *đường level set* là một *vết cắt* nếu ta cắt các bề mặt bởi một mặt phẳng song song với mặt phẳng $0xy$.

Khi khảo sát tính lồi của một hàm số hai biến, hoặc để tìm điểm cực trị của nó, người ta thường vẽ các level set thay vì vẽ các mặt trong không gian ba chiều. Hình 23.12 minh họa một vài ví dụ về các level set. Ở hàng trên, các *đường level set* là các đường khép kín. Khi các đường kín này tập trung nhỏ dần ở một điểm thì các điểm đó là các điểm cực trị. Với các hàm convex như trong ba ví dụ này, chỉ có một 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 biên của các tập lồi. Ở hàng dưới, các đường không phải khép kín. Hình 23.12d minh họa các level set của một hàm tuyến tính $f(x, y) = x + y$, và đó là một hàm *convex*. Hình 23.12e cũng minh họa các level set của một hàm lồi (chúng ta sẽ sớm thấy chứng minh) nhưng các level set là các *đường không kín*. Hàm này có chứa 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



Hình 23.12: Ví dụ về các level set. 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.

đương cũng là một *tập lỗi* vì nó là một polyhedron). Các *đường không kín* này nếu kết hợp với trục Ox, Oy sẽ tạo thành biên của các tập lỗi. Hình 23.12f minh họa các level set của một hàm hyperbolic, hàm này không phải là một hàm lỗi.

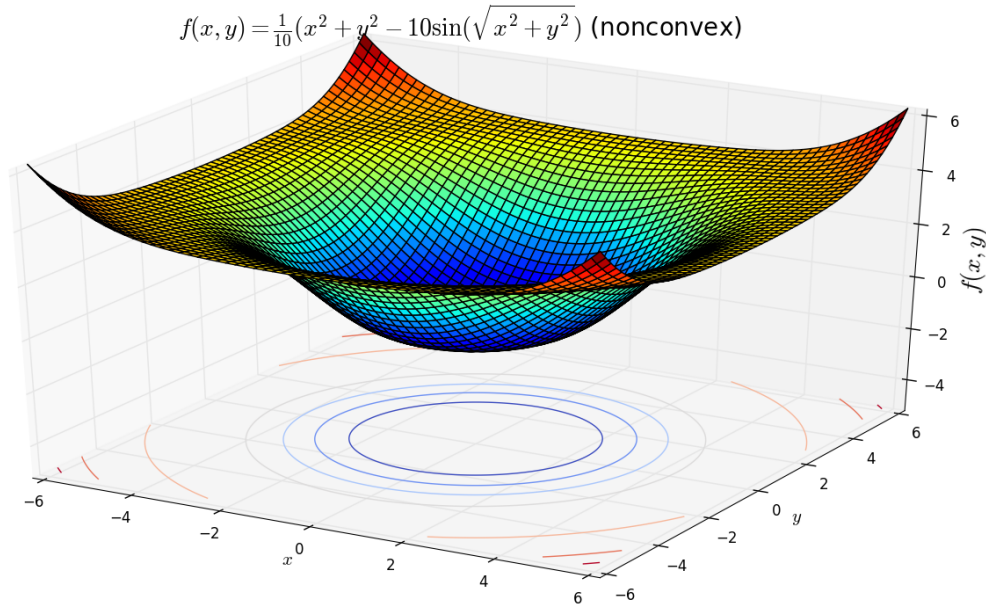
23.3.5 α -sublevel sets

Định nghĩa 23.8: α -sublevel set

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

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

Diễn đạt bằng lời, một α -sublevel set của một hàm số $f(\cdot)$ là tập hợp các điểm trong tập xác định của $f(\cdot)$ mà tại đó hàm số đạt giá trị không lớn hơn α .



Hình 23.13: Mọi α -sublevel sets là convex sets nhưng hàm số là nonconvex.

Quay lại với Hình 23.12, hàng trên, các α -sublevel sets chính là các hình lồi được bao bởi các level set. Ở Hình 23.12d, các α -sublevel sets chính là phần nửa mặt phẳng phía dưới xác định bởi các đường thẳng level set. Ở Hình 23.12e, các α -sublevel set chính là các vùng bị giới hạn bởi các trục tọa độ và các đường level set. Ở Hình 23.12f, các α -sublevel set hơi khó tưởng tượng một chút. Với $\alpha > 0$, các level sets là các đường màu vàng hoặc đỏ, các α -sublevel set tương ứng là phần nằm giữa các đường cùng màu. Các vùng này, có thể dễ nhận thấy, là *không lồi*.

Định lý 23.2

Nếu một hàm số là lồi thì mọi α -sublevel set của nó là lồi. Điều ngược lại chưa chắc đã đúng, tức nếu các α -sublevel set 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* (*nonconvex*), thì hàm số đó là *không lồi* (không lồi không có nghĩa là *concave*, chú ý). Vì vậy, hàm hyperbolic không phải là một hàm lồi. Các ví dụ ở Hình 23.12, trừ Hình 23.12f, đều tương ứng với các hàm lồi.

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

Hình 23.13 là một ví dụ khác về việc một hàm số có mọi α -sublevel set là lồi nhưng không phải là một hàm lồi. Mọi α -sublevel set của hàm số này đều là các hình tròn – lồi, 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. Chẳng hạn, đoạn thẳng nối một điểm ở *cánh* và một điểm ở *đáy* không nằm hoàn toàn phía trên của mặt.

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

Định nghĩa 23.9: 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 ở việc sử dụng hàm max.

23.3.6 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 bậc hai của nó. Tất nhiên là trong trường hợp các đạo hàm đó tồn tại.

First-order condition

Trước hết chúng ta định nghĩa phương trình 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, phương trình tiếp tuyến tại điểm co hoành độ $(x_0, f(x_0))$ là

$$y = f'(x_0)(x - x_0) + f(x_0)$$

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

Giả sử hàm số f có tập xác định là 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:

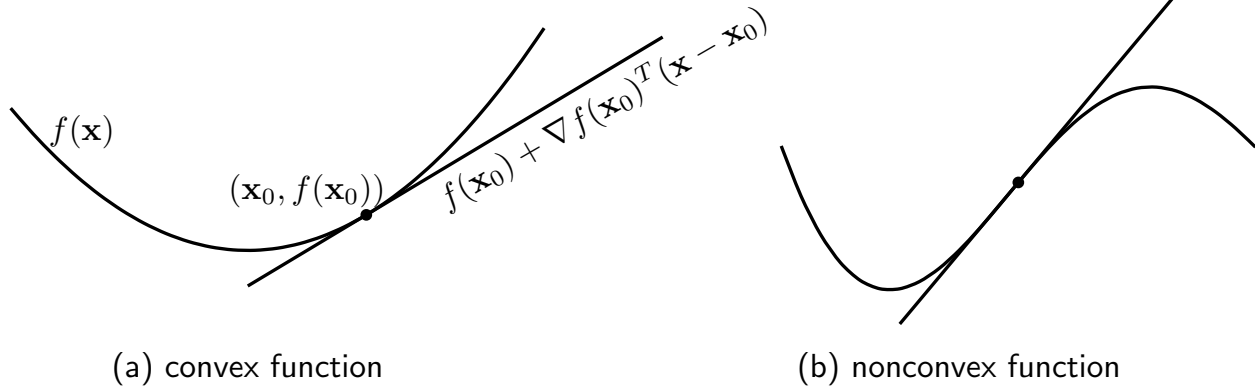
$$f(\mathbf{x}) \geq f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) \quad (23.3)$$

Tương tự như thế, một hàm số là *strictly convex* nếu dấu bằng trong (23.3) xảy ra khi và chỉ khi $\mathbf{x} = \mathbf{x}_0$.

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

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 23.14: 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.

Hình 23.14 minh họa đồ thị của một hàm lồi và một hàm không lồi. Hình 23.14a mô tả một hàm lồi. Hình 23.14b mô tả một hàm không lồi vì đồ thị của nó vừa nằm trên, vừa nằm dưới đường thẳng tiếp tuyến.

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 lồi.

Chứng minh: Đạo hàm bậc nhất của $f(\mathbf{x})$ 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 một hàm lồi. \square

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 $f(\mathbf{x})$, còn được gọi là *Hessian*, được ký hiệu là $\nabla^2 f(\mathbf{x})$.

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 của một hàm số là một ma trận *xác định dương* thì hàm số đó là *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 \mathbf{A} chính là Hessian của nó.
- 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*.

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 [BV04].

23.4 Tóm tắt

- Machine learning và tối ưu có quan hệ mật thiết với nhau. Trong tối ưu, tối ưu lồi là quan trọng nhất.
- Trong một tập lồi, 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 tập lồi là một tập lồi.
- Một hàm số là lồi nếu đoạn thẳng nối hai điểm bất kỳ trên đồ thị thì hàm số đó *không nằm dưới* đồ thị đó.
- Một hàm số khả vi là lồi nếu tập xác định của nó là lồi nếu mặt tiếp tuyến tại một điểm bất kỳ *không nằm phía trên* đồ thị của hàm số đó.
- Các norms là các hàm lồi, được sử dụng nhiều trong tối ưu.

Bài toán tối ưu lồi

24.1 Giới thiệu

Chúng ta cùng bắt đầu bài viết bằng ba bài toán tối ưu khá gần với thực tế.

24.1.1 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:

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

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: $x + z = 600$.

- 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 lớn. Với bài toán này, giả sử rằng x, y, z, t là các số thực dương. Nghiệm tìm được sẽ được làm tròn tới số tự nhiên gần nhất.

Vậy ta cần giải bài toán tối ưu sau đây:

Bài toán NXB¹

$$\begin{aligned}
 (x, y, z, t) &= \arg \min_{x, y, z, t} 5x + 10y + 15z + 4t \\
 \text{thoả mãn: } &x + z = 600 \\
 &y + t = 400 \\
 &x + y \leq 800 \\
 &z + t \leq 700 \\
 &x, y, z, t \geq 0
 \end{aligned} \tag{24.1}$$

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 siêu phẳng hoặc nửa không gian, đều là các ràng buộc tuyến tính (linear constraints). Bài toán tối ưu với cả objective function và constraints đều là tuyến tính được gọi là quy hoạch tuyến tính (**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 chương này.

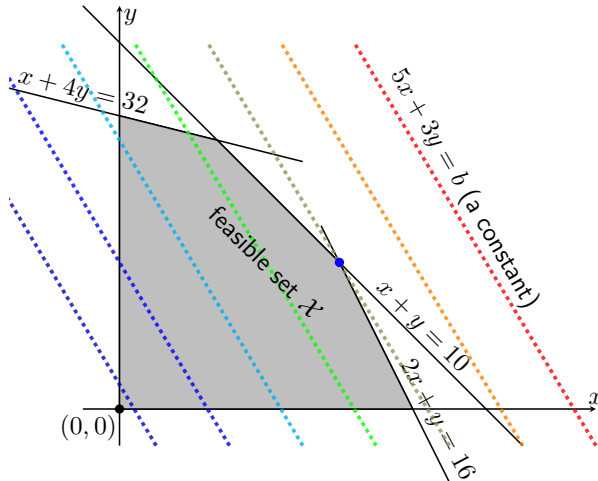
24.1.2 Bài toán canh tác

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 diện tích 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 quy hoạch như thế nào để tối đa lợi nhuận?

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). Đây chính là hàm mục tiêu của bài toán. Các ràng buộc trong bài toán này được viết dưới dạng:

¹ Nghiệm cho bài toán này có thể nhận thấy ngay là $x = 600, y = 0, z = 0, t = 400$. Nếu số lượng ràng buộc và số biến nhiều hơn, chúng ta cần một lời giải có thể tìm được bằng cách lập trình.



Hình 24.1: Minh họa nghiệm cho bài toán cạnh tác. 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.

- Tổng diện tích trồng không vượt quá 10ha: $x + y \leq 10$.
- 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 cạnh tác

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

$$\begin{aligned} \text{thoả mãn: } & x + y \leq 10 \\ & 2x + y \leq 16 \\ & x + 4y \leq 32 \\ & x, y \geq 0 \end{aligned} \tag{24.2}$$

Bài toán này yêu cầu *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 hàm mục tiêu. Khi đó hàm mục tiêu vẫn là tuyến tính, các ràng buộc là tuyến tính, ta lại có một bài toán **linear programming** nữa. Hình 24.1 minh họa nghiệm cho bài toán cạnh tác.

Vùng màu xám có dạng *polyhedron* (trong trường hợp này là đa giác) chính là tập hợp các điểm thoả mãn các ràng buộc. Các đường nét đứt có màu chính là các đường *đồng mức* (*level set*) của hàm mục tiêu $5x + 3y$, mỗi đường ứng với một giá trị khác nhau với màu 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. Trong khi đó, nếu trồng toàn bộ hồ tiêu trong 32 ngày, tức 8ha, anh chỉ thu được 24 triệu đồng.

Với các bài toán tối ưu có nhiều biến hơn và nhiều ràng buộc hơn, sẽ rất khó để minh họa và tìm nghiệm như cách này. Chúng ta cần có một công cụ hiệu quả hơn, tốt nhất là nghiệm có thể tìm được bằng cách lập trình.

24.1.3 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 $2T/m^2$. Hỏi 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 chứa 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, chiều rộng, chiều cao lần lượt là x, y, z (m). Thể tích của thùng là xyz (đơn vị là m^3). Có hai loại chi phí:

- *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 $0.1 \frac{400}{xyz} = \frac{40}{xyz} = 40x^{-1}y^{-1}z^{-1}$ (0.1 ở đây là 0.1 triệu đồng).
- *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 đây.

Bài toán vận chuyển:

$$(x, y) = \arg \min_{x, y, z} 40x^{-1}y^{-1}z^{-1} + 2(xy + yz + zx) \quad (24.3)$$

thỏa mãn: $x, y, z > 0$

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 chương.

Bảng 24.1: Bảng các thuật ngữ và ký hiệu trong các bài toán tối ưu.

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

Nhận thấy rằng bài này hoàn toàn có thể dùng bất đẳng thức Cauchy để giải được, nhưng chúng ta 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 $x = y = z = \sqrt[5]{10}$.)

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 của chương. Trước hết, chúng ta cần hiểu các khái niệm về các bài toán tối ưu lồi *convex optimization problems* và tại sao chúng lại quan trọng.

24.2 Nhắc lại bài toán tối ưu

24.2.1 Các khái niệm cơ bản

Bài toán tối ưu ở dạng tổng quát:

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thoả mãn: } & f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ & h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned} \tag{24.4}$$

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 khái niệm và ký hiệu trong bài toán tối ưu bằng cả tiếng Anh và tiếng Việt như trong Bảng 24.1. Ngoài ra,

- Khi $m = p = 0$, bài toán (24.4) được gọi là *bài toán tối ưu không ràng buộc* (*unconstrained optimization problem*).
- \mathcal{D} 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 (24.4) là *infeasible* (vô nghiệm). Nếu một điểm nằm trong *feasible set*, ta gọi điểm đó là *feasible*.
- *Optimal value* (giá trị tối ưu) của bài toán tối ưu (24.4) được định nghĩa là:

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

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

24.2.2 Optimal và locally optimal points

Một điểm \mathbf{x}^* được gọi là một điểm *optimal point* (điểm tối ưu), hoặc là *nghiệm* của bài toán (24.4) nếu \mathbf{x}^* là *feasible* và $f_0(\mathbf{x}^*) = p^*$. Tập hợp tất cả các *optimal point* được gọi là *optimal set*. Nếu *optimal set* là một tập không rỗng, ta nói bài toán (24.4) là *giải được* (*solvable*). 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 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 một chiều, *lân cận* của một điểm được hiểu là tập các điểm cách điểm đó một khoảng rất nhỏ. Trong không gian nhiều chiều, ta gọi một điểm \mathbf{x} là *locally optimal* nếu tồn tại một giá trị $R > 0$ sao cho:

$$f_0(\mathbf{x}) = \inf \{f_0(\mathbf{z}) \mid f_i(\mathbf{z}) \leq 0, i = 1, \dots, m, h_j(\mathbf{z}) = 0, j = 1, \dots, p, \|\mathbf{z} - \mathbf{x}\|_2 \leq R\} \quad (24.5)$$

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} .

24.2.3 Một vài lưu ý

Mặc dù trong định nghĩa bài toán tối ưu (24.4) 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 $f_i(\mathbf{x}) \leq 0$.

24.3 Bài toán tối ưu lỗi

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.

24.3.1 Định nghĩa

Định nghĩa 24.1: Bài toán tối ưu lỗi

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

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thỏa mãn: } & f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ & h_j(\mathbf{x}) = \mathbf{a}_j^T \mathbf{x} - b_j = 0, \quad j = 1, \dots, \end{aligned}$$

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

So với bài toán tối ưu (24.4), bài toán tối ưu lỗi (24.6) 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*.

Một vài nhận xét:

- Tập hợp các điểm thỏa 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 thỏa 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 lỗi, vì vậy nó là một tập lỗi.

Trong bài toán tối ưu lỗi, ta tối thiểu một hàm mục tiêu lỗi trên một tập lỗi.

24.3.2 Local optimum của bài toán tối ưu lỗi chính là global optimum của nó

Tính chất quan trọng nhất của bài toán tối ưu lỗi chính là mọi điểm *locally optimal point* chính là một điểm (*globally*) *optimal point* (điểm cực tiểu chính là nghiệm của bài toán). Việc này có thể chứng minh bằng phản chứng.. Gọi \mathbf{x}_0 là một điểm *locally optimal*:

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

với $R > 0$ nào đó. Giả sử \mathbf{x}_0 không phải là một điểm *globally optimal*, tức tồn tại một điểm *feasible* \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 $\|\mathbf{z} - \mathbf{x}_0\|_2 < R$. Việc này có được vì feasible set là một tập lỗi. Hơn nữa, vì hàm mục tiêu f_0 là một hàm lồi, ta có

$$f_0(\mathbf{z}) = f_0((1 - \theta)\mathbf{x}_0 + \theta\mathbf{y}) \tag{24.6}$$

$$\leq (1 - \theta)f_0(\mathbf{x}_0) + \theta f_0(\mathbf{y}) \tag{24.7}$$

$$< (1 - \theta)f_0(\mathbf{x}_0) + \theta f_0(\mathbf{x}_0) = f_0(\mathbf{x}_0) \tag{24.8}$$

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. \square

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 lỗi, điểm cực tiểu phải là điểm làm cho hàm số đạt giá trị nhỏ nhất.

24.3.3 Điều kiện tối ưu cho hàm mục tiêu khả vi

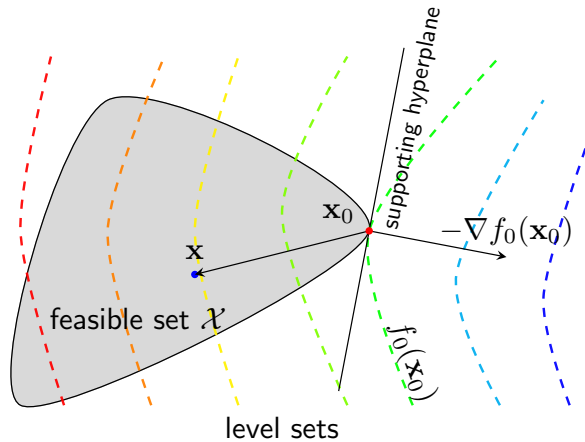
Nếu hàm mục tiêu f_0 là khả vi, theo first-order condition, với mọi $\mathbf{x}, \mathbf{y} \in \text{dom} f_0$, ta có:

$$f_0(\mathbf{x}) \geq f_0(\mathbf{x}_0) + \nabla f_0(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \tag{24.9}$$

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

$$\nabla f_0(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \geq 0, \forall \mathbf{x} \in \mathcal{X} \tag{24.10}$$

Phần chứng minh cho điều kiện này được bỏ qua, bạn đọc có thể tìm trong trang 139-140 của cuốn Convex Optimization [BV04].



Hình 24.2: 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 chỉ ra rằng nếu $\nabla f_0(\mathbf{x}_0) = 0$ thì \mathbf{x}_0 chính là một điểm optimal của bài toán. Nếu $\nabla f_0(\mathbf{x}_0) \neq 0$, nghiệm của bài toán sẽ phải nằm trên biên của feasible set. Thật vậy, quan sát Hình 24.2, điều kiện này nói rằng nếu \mathbf{x}_0 là một 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. 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. Điều này chỉ ra rằng \mathbf{x}_0 phải nằm trên biên của feasible set \mathcal{X} . 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, chúng ta dùng màu lam để chỉ giá trị nhỏ, màu đỏ để chỉ giá trị lớn của hàm.

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

24.3.4 Giới thiệu thư viện CVXOPT

CVXOPT là một thư viện miễn phí trên Python đi kèm với cuốn sách Convex Optimization. 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 (<http://cvxopt.org/>). Trong phần còn lại của chương, chúng ta sẽ thảo luận ba bài toán cơ bản trong convex optimization: linear programming, quadratic programming, và geometric programming. Chúng ta sẽ cùng lập trình để giải các ví dụ đã nêu ở phần đầu bài viết dựa trên thư viện CVXOPT này.

24.4 Linear programming

Chúng ta cùng bắt đầu với lớp các bài toán đơn giản nhất trong convex optimization - linear programming (LP). Trong đó, hàm mục tiêu $f_0(\cdot)$ và các hàm bất đẳng thức ràng buộc $f_i(\cdot), i = 1, \dots, m$ đều là các hàm *affine*.

24.4.1 Dạng tổng quát của linear programming

Dạng tổng quát (general form) của linear programming

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} + d \\ \text{thoả mãn: } \quad \mathbf{G}\mathbf{x} &\preceq \mathbf{h} \\ \mathbf{A}\mathbf{x} &= \mathbf{b} \end{aligned} \tag{24.11}$$

Trong đó $\mathbf{G} \in \mathbb{R}^{m \times n}$, $\mathbf{h} \in \mathbb{R}^m$, $\mathbf{A} \in \mathbb{R}^{p \times n}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$ và $d \in \mathbb{R}$.

Số vô hướng d chỉ làm thay đổi giá trị của hàm mục tiêu mà không làm thay đổi nghiệm của bài toán nên có thể được lược bỏ. Nhắc lại rằng ký hiệu \preceq nghĩa là mỗi phần tử trong vector ở vế trái nhỏ hơn hoặc bằng phần tử tương ứng trong vector ở vế 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 $\mathbf{G}\mathbf{x} \preceq \mathbf{h}$ 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 .

24.4.2 Dạng tiêu chuẩn của linear programming

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

Dạng tiêu chuẩn của linear programming

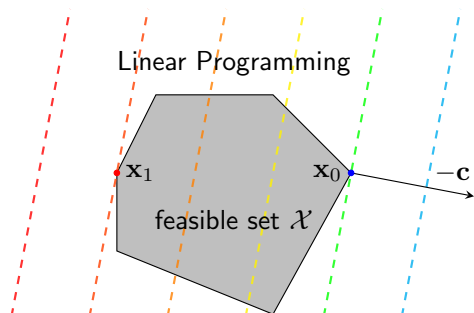
$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \\ \text{thoả mãn: } \quad \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\succeq \mathbf{0} \end{aligned} \tag{24.12}$$

Dạng tổng quát (24.11) có thể được đưa về dạng tiêu chuẩn (24.12) bằng cách đặt thêm biến *slack* \mathbf{s} .

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}, \mathbf{s}} \mathbf{c}^T \mathbf{x} \\ \text{thoả mãn: } \quad \mathbf{A}\mathbf{x} &= \mathbf{b} \\ \mathbf{G}\mathbf{x} + \mathbf{s} &= \mathbf{h} \\ \mathbf{s} &\succeq \mathbf{0} \end{aligned} \tag{24.13}$$

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 (24.13) dưới dạng:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}^+, \mathbf{x}^-, \mathbf{s}} \mathbf{c}^T \mathbf{x}^+ - \mathbf{c}^T \mathbf{x}^- \\ \text{thoả mãn: } \quad \mathbf{A}\mathbf{x}^+ - \mathbf{A}\mathbf{x}^- &= \mathbf{b} \\ \mathbf{G}\mathbf{x}^+ - \mathbf{G}\mathbf{x}^- + \mathbf{s} &= \mathbf{h} \\ \mathbf{x}^+ \succeq \mathbf{0}, \mathbf{x}^- \succeq \mathbf{0}, \mathbf{s} &\succeq \mathbf{0} \end{aligned} \tag{24.14}$$



Hình 24.3: Biểu diễn hình học của linear programming.

Tới đây, bạn đọc có thể thấy rằng (24.14) có thể viết gọn lại như (24.12).

24.4.3 Minh hoạ bằng hình học của bài toán linear programming

Các bài toán LP có thể được minh hoạ như Hình 24.3. 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. Nghiệm của các bài toán LP, nếu có, thường là một điểm ở *đỉnh* của polyhedron feasible set 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).

Tiếp theo, chúng ta sẽ dùng thư viện CVXOPT để giải các bài toán LP.

24.4.4 Giải LP bằng CVXOPT

Nhắc lại bài toán cạnh tác

$$\begin{aligned} (x, y) &= \arg \max_{x, y} 5x + 3y \\ \text{thoả mãn: } &x + y \leq 10 \\ &2x + y \leq 16 \\ &x + 4y \leq 32 \\ &x, y \geq 0 \end{aligned} \tag{24.15}$$

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

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

Khi sử dụng CVXOPT, chúng ta lập trình như sau:

```

from cvxopt import matrix, solvers
c = matrix([-5., -3.]) # since we need to maximize the objective function
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'])

```

Kết quả:

```

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

```

Nghiệm này chính là nghiệm mà chúng ta đã tìm được trong phần đầu của bài viết dựa trên biểu diễn hình học.

Một vài lưu ý:

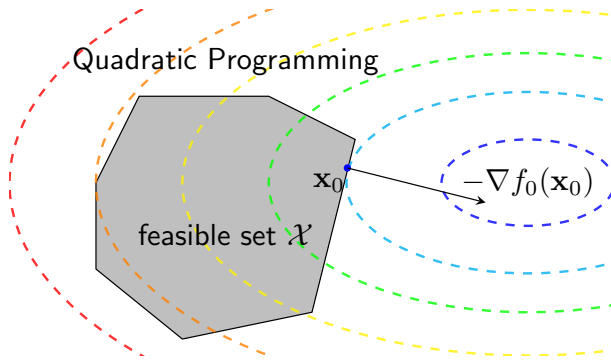
- Hàm `solvers.lp` của `cvxopt` giải bài toán (24.13).
- Trong bài toán của chúng ta, vì ta cần tìm giá trị lớn nhất nên ta phải đổi hàm mục tiêu về dạng $-5x - 3y$. Chính vì vậy mà `c = matrix([-5., -3.])`.
- Hàm `matrix` nhận đầu vào là một `list` (trong Python), `list` này thể hiện một vector cột. 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 thể thể hiện đó là số thực.
- 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 (<https://goo.gl/q5CZmz>). Việc giải Bài toán NXB bằng CVXOPT xin nhường lại cho bạn đọc.

24.5 Quadratic programming

24.5.1 Bài toán quadratic programming

Một dạng bài toán convex optimization phổ biến khác là *quadratic programming* (QP). Khác biệt duy nhất của QP so với LP là hàm mục tiêu có dạng *toàn phương* (*quadratic form*).



Hình 24.4: Biểu diễn hình học của Quadratic Programming.

Quadratic Programming

$$\mathbf{x} = \arg \min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + \mathbf{r} \quad (24.16)$$

thoả mãn: $\mathbf{G} \mathbf{x} \preceq \mathbf{h}$
 $\mathbf{A} \mathbf{x} = \mathbf{b}$

Trong đó \mathbf{P} là một ma trận vuông nửa xác định dương bậc n , $\mathbf{G} \in \mathbb{R}^{m \times n}$, $\mathbf{A} \in \mathbb{R}^{p \times n}$.

Điều kiện nửa xác định dương của \mathbf{P} để đảm bảo rằng hàm mục tiêu là convex. Trong QP, một hàm quadratic lồi được tối thiểu trên một *polyhedron* (Xem Hình 24.4). LP chính là một trường hợp đặc biệt của QP với $\mathbf{P} = \mathbf{0}$.

24.5.2 Ví dụ về QP

Bài toán vui: Có một hòn đảo 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. Có thể thấy rằng nghiệm của bài toán chính là một góc của đảo gần con thuyền nhất hoặc hình chiếu vuông góc của thuyền tới cạnh gần nhất của đảo. Đây chính là bài toán tìm khoảng cách từ một điểm tới một polyhedron.

Bài toán tìm khoảng cách từ một điểm tới một polyhedron: cho một polyhedron là tập hợp các điểm thoả mãn $\mathbf{A} \mathbf{x} \preceq \mathbf{b}$, 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. Đây là một bài toán QP có dạng

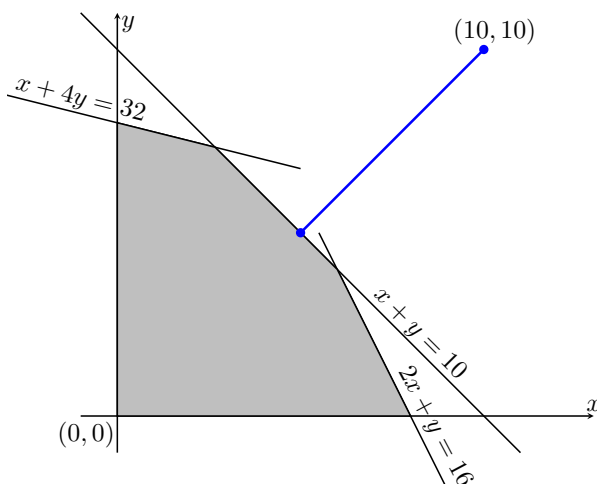
$$\mathbf{x} = \arg \min_{\mathbf{x}} \frac{1}{2} \|\mathbf{x} - \mathbf{u}\|_2^2$$

thoả mãn: $\mathbf{G} \mathbf{x} \preceq \mathbf{h}$

Hàm mục tiêu đạt giá trị nhỏ nhất bằng 0 nếu \mathbf{u} nằm trong polyhedron đó và nghiệm 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 (24.16) 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.



Hình 24.5: Ví dụ về khoảng cách giữa một điểm và một polyhedron.

24.5.3 Giải QP bằng CVXOPT

Xét bài toán được cho trên Hình 24.5. Ta cần tìm khoảng cách từ điểm có tọa độ $(10, 10)$ tới hình đa giác lồi màu xám. Chú ý rằng khoảng cách từ một điểm tới một tập hợp chính là khoảng cách từ điểm đó tới điểm gần nhất trong tập hợp. Bài toán này được viết dưới dạng QP như sau:

$$(x, y) = \arg \min_{x, y} (x - 10)^2 + (y - 10)^2$$

thoả mãn:

$$\begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 4 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \preceq \begin{bmatrix} 10 \\ 16 \\ 32 \\ 0 \\ 0 \end{bmatrix}$$

Feasible set trong bài toán được lấy 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'])
```


Kết quả:

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

Như vậy, nghiệm của bài toán tối ưu này là điểm có tọa độ (5, 5) .

24.6 Geometric Programming

Trong mục này, chúng ta cùng thảo luận một lớp các bài toán *không lỗi* khi quan sát hàm mục tiêu và các hàm ràng buộc, nhưng có thể được biến đổi về dạng *lỗi* bằng một vài kỹ thuật không quá phức tạp. Trước hết, ta làm quen với hai khái niệm *monomial* và *posynomial*.

24.6.1 Monomial và posynomial

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

$$f(\mathbf{x}) = cx_1^{a_1} x_2^{a_2} \dots x_n^{a_n} \quad (24.17)$$

trong đó $c > 0$ và $a_i \in \mathbb{R}$, được gọi là một *monomial function* (khái niệm này khá giống với *đơn thức* trong chương trình phổ thông, nhưng sách giáo khoa định nghĩa với c bất kỳ và a_i là các số tự nhiên).

Tổng của các monomial

$$f(\mathbf{x}) = \sum_{k=1}^K c_k x_1^{a_{1k}} x_2^{a_{2k}} \dots x_n^{a_{nk}} \quad (24.18)$$

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

24.6.2 Geometric programming

Geometric programming (GP)

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thỏa mãn: } f_i(x) &\leq 1, \quad i = 1, 2, \dots, m \\ h_j(x) &= 1, \quad j = 1, 2, \dots, p \end{aligned} \quad (24.19)$$

trong đó f_0, f_1, \dots, f_m là các *posynomials* và h_1, \dots, h_p là các *monomials*.

Điều kiện $\mathbf{x} \succ 0$ đã được ẩn đi.

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

Ví dụ, bài toán tối ưu

$$\begin{aligned} (x, y, z) &= \arg \min_{x, y, z} x/y \\ \text{thoả mãn: } & 1 \leq x \leq 2 \\ & x^3 + 2y/z \leq \sqrt{y} \\ & x/y = z \end{aligned} \tag{24.20}$$

có thể được viết lại dưới dạng GP:

$$\begin{aligned} (x, y, z) &= \arg \min_{x, y, z} xy^{-1} \\ \text{thoả mãn: } & x^{-1} \leq 1 \\ & (1/2)x \leq 1 \\ & x^3y^{-1/2} + 2y^{1/2}z^{-1} \leq 1 \\ & xy^{-1}z^{-1} = 1 \end{aligned} \tag{24.21}$$

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

24.6.3 Biến đổi GP về dạng bài toán tối ưu lồi

GP có thể được biến đổi về dạng lồi bằng cách sau đây. Đặt $y_i = \log(x_i)$, tức $x_i = \exp(y_i)$. Nếu f là một *monomial function* của \mathbf{x} thì:

$$f(\mathbf{x}) = c(\exp(y_1))^{a_1} \dots (\exp(y_n))^{a_n} = c \exp\left(\sum_{i=1}^n a_i y_i\right) = \exp(\mathbf{a}^T \mathbf{y} + b)$$

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 (24.18) có thể được viết dưới dạng

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

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

Bài toán GP (24.19) được viết lại dưới dạng:

$$\begin{aligned} \mathbf{y} &= \arg \min_{\mathbf{y}} \sum_{k=1}^{K_0} \exp(\mathbf{a}_{0k}^T \mathbf{y} + b_{0k}) \\ \text{thoả mãn: } & \sum_{k=1}^{K_i} \exp(\mathbf{a}_{ik}^T \mathbf{y} + b_{ik}) \leq 1, \quad i = 1, \dots, m \\ & \exp(\mathbf{g}_j^T \mathbf{y} + h_j) = 1, \quad j = 1, \dots, p \end{aligned} \tag{24.22}$$

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

Với chú ý rằng hàm số $\log(\sum_{i=1}^m \exp(g_i(\mathbf{z})))$ là một hàm *lồi* theo \mathbf{z} nếu g_i là các hàm *lồi* (xin bỏ qua phần chứng minh), ta có thể viết lại bài toán (24.22) dưới dạng *lồi* bằng cách lấy log của các hàm như sau.

Geometric programming dưới dạng bài toán tối ưu lồi

$$\begin{aligned} \text{minimize}_{\mathbf{y}} \tilde{f}_0(\mathbf{y}) &= \log \left(\sum_{k=1}^{K_0} \exp(\mathbf{a}_{0k}^T \mathbf{y} + b_{i0}) \right) \\ \text{thỏa mãn: } \tilde{f}_i(\mathbf{y}) &= \log \left(\sum_{k=1}^{K_i} \exp(\mathbf{a}_{ik}^T \mathbf{y} + b_{ik}) \right) \leq 0, \quad i = 1, \dots, m \\ \tilde{h}_j(\mathbf{y}) &= \mathbf{g}_j^T \mathbf{y} + h_j = 0, \quad j = 1, \dots, p \end{aligned} \tag{24.23}$$

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 (24.23) 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 dạng định nghĩa của GP).

24.6.4 Giải GP bằng CVXOPT

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.

Nghiệm của bài toán có thể được tìm 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] # number of monomials
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)
```

Kết quả:

```
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` (<https://goo.gl/5FEBtn>) để hiểu cách thiết lập và giải bài toán GP.

24.7 Tóm tắt

- Các bài toán tối ưu xuất hiện rất nhiều trong thực tế, trong đó tối ưu lỗi đóng một vai trò quan trọng. Trong bài toán tối ưu lỗi, 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 lỗi nhưng có thể biến đổi về dạng lỗi, 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 lỗi, rất dễ sử dụng và thời gian chạy tương đối nhanh. Phù hợp với mục đích học tập và nghiên cứu.

Duality

25.1 Giới thiệu

Trong Chương 23, chúng ta đã làm quen với các khái niệm về tập hợp lồi và hàm số lồi. Tiếp theo đó, trong Chương 24, chúng ta đã thảo luận các bài toán tối ưu lồi, cách nhận dạng và cách sử dụng thư viện để giải các bài toán tối ưu lồi cơ bản. Trong chương 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 xét bài toán mà ràng buộc chỉ là một phương trình:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thỏa mãn: } f_1(\mathbf{x}) &= 0 \end{aligned} \tag{25.1}$$

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. Bài toán này có thể được giải bằng phương pháp nhân tử Lagrange (xem Phụ Lục A). Cụ thể, xét 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 (25.1) thỏa mãn điều kiện $\nabla_{\mathbf{x}, \lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$. Điều này tương đương với

$$\nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \tag{25.2}$$

$$f_1(\mathbf{x}) = 0 \tag{25.3}$$

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 (25.1). Việc giải hệ phương trình (25.2) - (25.3), trong nhiều trường hợp, đơn giản hơn việc trực tiếp đi tìm *optimal value* của bài toán (25.1). Một vài ví dụ về phương pháp nhân tử Lagrange có thể được tìm thấy tại Phụ Lục A.

25.2 Hàm đối ngẫu Lagrange

25.2.1 Lagrangian

Với bài toán tối ưu tổng quát

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thoả mãn: } & f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ & h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned} \quad (25.4)$$

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). Bài toán tối ưu này còn được gọi là *bài toán chính* (*primal problem*).

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 *biến đối ngẫu* (*dual variables*) hoặc *vector nhân tử Lagrange* (*Lagrange multiplier vectors*). Lúc này nếu biến chính $\mathbf{x} \in \mathbb{R}^n$ thì tổng số biến của hàm số này sẽ là $n + m + p$.

25.2.2 Hàm đối ngẫu Lagrange

Hàm đối ngẫu Lagrange (*the Lagrange dual function*) của bài toán tối ưu (hoặc gọn là *hàm số đối ngẫu*) (25.4) là một hàm của các biến đối ngẫu $\boldsymbol{\lambda}$ và $\boldsymbol{\nu}$, được định nghĩa là giá trị nhỏ nhất theo \mathbf{x} của *Lagrangian*:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x} \in \mathcal{D}} \left(f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j h_j(\mathbf{x}) \right) \quad (25.5)$$

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* – là tập hợp các điểm thoả mãn các ràng buộc. *Feasible set* là một 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 vừa *convex*, vừa *concave*. Vậy, hàm đối ngẫu chính là một *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* (xem thêm Mục 23.3.2).

25.2.3 Chặn dưới của giá trị tối ưu

Nếu p^* là optimal value (giá trị tối ưu) của bài toán (25.4) thì với các biến đối ngẫu $\lambda_i \geq 0, \forall i$ và ν bất kỳ, chúng ta sẽ có

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \leq p^* \quad (25.6)$$

Tính chất này có thể được chứng minh như sau. Giả sử \mathbf{x}_0 là một điểm *feasible* bất kỳ của bài toán (25.4), 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

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

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 $\mathbf{x}_0 = \mathbf{x}^*$ (optimal point), $f_0(\mathbf{x}_0) = p^*$, ta suy ra bất đẳng thức (25.6). Bất đẳng thức quan trọng này chỉ ra rằng giá trị tối ưu của hàm mục tiêu trong dual problem (25.4) không nhỏ hơn giá trị lớn nhất của hàm đối ngẫu Lagrange $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$.

25.2.4 Ví dụ

Ví dụ 1: Xét bài toán tối ưu

$$x = \arg \min_x x^2 + 10 \sin(x) + 10 \quad (25.7)$$

$$\text{thoả mãn: } (x - 2)^2 \leq 4$$

Với bài toán này, miền xác định $\mathcal{D} = \mathbb{R}$ nhưng *feasible set* là $0 \leq x \leq 4$. Đồ thị của hàm mục tiêu được minh hoạ bởi đường đậm màu lam trong Hình 25.1a. Hàm số ràng buộc $f_1(x) = (x - 2)^2 - 4$ được cho bởi đường nét đứt màu lục. Optimal value của bài toán này có thể được nhận ra là điểm trên đồ thị có hoành độ bằng 0 (là điểm nhỏ nhất trên đường màu lam trong đoạn $[0, 4]$). 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.

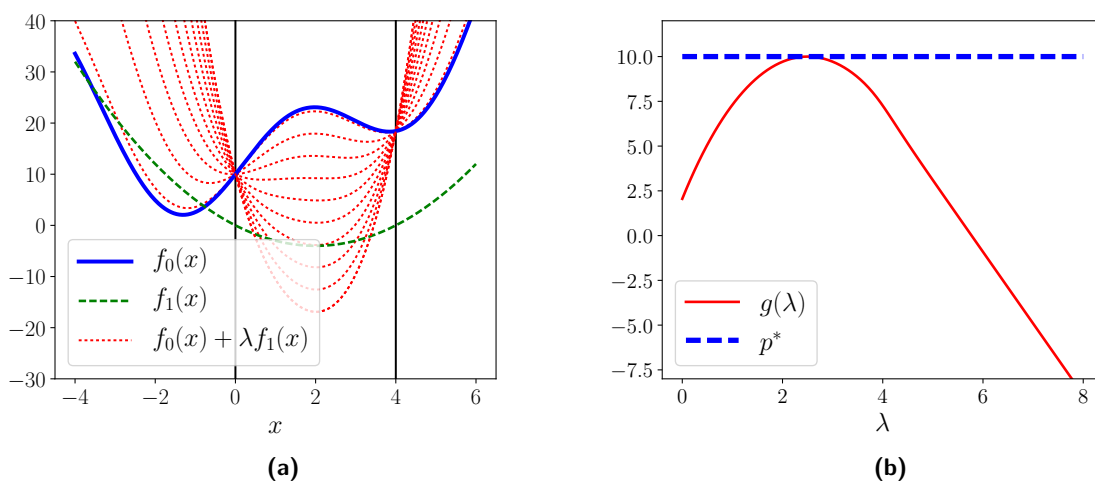
Lagrangian của bài toán này có dạng

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

Các đường dấu chấm màu đỏ trong Hình 25.1a 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:

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



Hình 25.1: Ví dụ về dual function. (a) Đường màu lam đậm thể hiện hàm mục tiêu. Đường nét đứt mà lục thể hiện hàm số ràng buộc. Các đường nét đứt màu đỏ thể hiện dual function ứng với các λ khác nhau. (b) 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 (source code cho hình vẽ này có thể được tìm thấy tại <https://goo.gl/jZiRCp>).

Từ Hình 25.1a, ta có thể thấy ngay rằng với các λ khác nhau, giá của $g(\lambda)$ hoặc tại điểm có hoành độ bằng 0 của đường màu lam, hoặc tại một điểm thấp hơn điểm đó. Đồ thị của hàm $g(\lambda)$ được cho bởi đường liền màu đỏ ở Hình 25.1b. 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 chiều trên-dưới thì đạt được đồ thị của một hàm convex.

Source code cho Hình 25.1 có thể được tìm thấy tại <https://goo.gl/jZiRCp>.

Ví dụ 2 Xét một bài toán linear programming:

$$\begin{aligned}
 x &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \\
 \text{thoả mãn: } \mathbf{Ax} &= \mathbf{b} \\
 \mathbf{x} &\succeq 0
 \end{aligned}
 \tag{25.8}$$

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 của nó là

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = -\mathbf{b}^T \boldsymbol{\nu} + \inf_{\mathbf{x}} (\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda})^T \mathbf{x} \quad (25.9)$$

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,

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \begin{cases} -\mathbf{b}^T \boldsymbol{\nu} & \text{nếu } \mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0 \\ -\infty & \text{o.w.} \end{cases} \quad (25.10)$$

Trường hợp thứ hai khi $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = -\infty$ chúng ta 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 $\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0$.

25.3 Bài toán đối ngẫu Lagrange

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 (25.4). 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

$$\boldsymbol{\lambda}^*, \boldsymbol{\nu}^* = \arg \max_{\boldsymbol{\lambda}, \boldsymbol{\nu}} g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \quad (25.11)$$

thoả mãn: $\boldsymbol{\lambda} \succeq 0$

Quan trọng, vì hàm $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ là *concave* và hàm ràng buộc $f_i(\boldsymbol{\lambda}) = -\lambda_i$ là các hàm *convex*. Vậy bài toán (25.11) chính là một bài toán *convex*. Vì vậy trong nhiều trường hợp, lời giải có thể dễ tìm hơn là bài toán gốc. Chú ý rằng, bài toán tối ưu này là *convex* bất kể bài toán gốc (25.4) có là *convex* hay không.

Bài toán tối ưu này được gọi là *bài toán đối ngẫu Lagrange (Lagrange dual problem)* ứng với bài toán chính (25.4). Ngoài ra, có một khái niệm nữa được 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 $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) > -\infty$ (điều kiện này được thêm vào vì ta chỉ quan tâm tới các $(\boldsymbol{\lambda}, \boldsymbol{\nu})$ sao cho hàm mục tiêu của bài toán đối ngẫu càng lớn càng tốt). Nghiệm của bài toán đối ngẫu (25.11), được 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*.

25.3.1 Weak duality

Ký hiệu giá trị tối ưu của bài toán đối ngẫu (25.11) là d^* . Theo (25.6), ta đã biết rằng $d^* \leq p^*$. Tính chất đơn giản này được gọi là *weak duality*. Tuy đơn giản nhưng nó cực kỳ quan trọng.

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 hàm mục tiêu trong bài toán đối ngẫu 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 (lỗi 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 khả thi vì bài toán đối ngẫu luôn luôn là lỗi.

25.3.2 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 lỗi, tức có dạng

$$\begin{aligned} x &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thoả mãn: } & f_i(\mathbf{x}) \leq 0, i = 1, 2, \dots, m \\ & \mathbf{Ax} = \mathbf{b} \end{aligned} \quad (25.12)$$

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 25.1: Strictly feasible

Một điểm *feasible* của bài toán (25.12) được gọi là *strictly feasible* nếu:

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

tức các dấu bằng trong các bất đẳng thức ràng buộc không xảy ra.

Định lý 25.1: Slater

Nếu tồn tại một điểm *strictly feasible* (bài toán gốc là lỗi) 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 25.1 phía trên.

25.4 Các điều kiện tối ưu

25.4.1 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

$$f_0(\mathbf{x}^*) = g(\lambda^*, \nu^*) \quad (25.13)$$

$$= \inf_{\mathbf{x}} \left(f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}) \right) \quad (25.14)$$

$$\leq f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}^*) \quad (25.15)$$

$$\leq f_0(\mathbf{x}^*) \quad (25.16)$$

Đẳng thức (25.13) xảy ra do *strong duality*. Đẳng thức (25.14) xảy ra do định nghĩa của hàm đối ngẫu. Bất đẳng thức (25.15) là hiển nhiên vì infimum của một hàm nhỏ hơn giá trị của hàm đó tại bất kỳ một điểm nào khác. Bất đẳng thức (25.16) xảy ra 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 (25.15) và (25.16) 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 $\lambda_i^* \geq 0, f_i \leq 0$ nên mỗi phần tử $\lambda_i^* f_i(\mathbf{x}^*) \leq 0$. Từ đó ta phải có $\lambda_i^* f_i(\mathbf{x}^*) = 0, \forall i = 1, 2, \dots, m$.

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

$$\lambda_i^* > 0 \Rightarrow f_i(\mathbf{x}^*) = 0 \quad (25.17)$$

$$f_i(\mathbf{x}^*) < 0 \Rightarrow \lambda_i^* = 0 \quad (25.18)$$

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

25.4.2 Các điều kiện tối ưu KKT

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

Điều kiện KKT cho bài toán *không* lồi

Giả sử rằng *strong duality* xảy ra. Gọi \mathbf{x}^* và $(\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*)$ là *bất kỳ 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 các điều kiện

$$f_i(\mathbf{x}^*) \leq 0, i = 1, 2, \dots, m \quad (25.19)$$

$$h_j(\mathbf{x}^*) = 0, j = 1, 2, \dots, p \quad (25.20)$$

$$\lambda_i^* \geq 0, i = 1, 2, \dots, m \quad (25.21)$$

$$\lambda_i^* f_i(\mathbf{x}^*) = 0, i = 1, 2, \dots, m \quad (25.22)$$

$$\nabla_{\mathbf{x}} f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* \nabla_{\mathbf{x}} f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* \nabla_{\mathbf{x}} h_j(\mathbf{x}^*) = 0 \quad (25.23)$$

Đây là *điều kiện cần* để $\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ là nghiệm của *primal problem* và *dual problem*.

Các điều kiện KKT 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 bộ $(\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*)$ nào thoả mãn các điều kiện KKT đều là *primal và dual optimal* của *primal problem* và *dual problem*.

Cần nhớ

Với một bài toán lồi và điều kiện Slater thoả mãn (suy ra *strong duality*) thì các điều kiện KKT là các điều 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 Phần Support Vector Machine), việc giải hệ (bất) phương trình các điều kiện KKT là khả thi. Rất nhiều các thuật toán tối ưu được xây dựng giả trên việc giải hệ điều kiện KKT.

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

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + r \\ \text{thoả mãn: } \quad & \mathbf{A} \mathbf{x} = \mathbf{b} \end{aligned} \quad (25.24)$$

trong đó \mathbf{P} là một ma trận nửa xác định dương. Lagrangian của bài toán này là

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

Điều kiện KKT cho bài toán này là:

$$\mathbf{A}\mathbf{x}^* = \mathbf{b} \quad (25.25)$$

$$\mathbf{P}\mathbf{x}^* + \mathbf{q} + \mathbf{A}^T\boldsymbol{\nu}^* = 0 \quad (25.26)$$

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 dưới dạng

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

Đây là một phương trình tuyến tính đơn giản!

25.5 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 $\boldsymbol{\lambda}, \boldsymbol{\nu}$. Với các giá trị $(\boldsymbol{\lambda}, \boldsymbol{\nu})$ cố định, ta có định nghĩa về **hàm đối ngẫu Lagrange** (Lagrange dual function) $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ được xác định là infimum của Lagrangian khi \mathbf{x} thay đổi trên miền xác định của bài toán.
- *Feasible set* là tập con của *domain set* (tập xác định).
- Với mọi $(\boldsymbol{\lambda}, \boldsymbol{\nu})$, $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \leq p^*$.
- Hàm số $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ là *convex* bất kể bài toán tối ưu gốc có *convex* 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 **convex** bất kể bài toán gốc có *convex* hay không.
- Gọi giá trị tối ưu của bài toán đối ngẫu là d^* , 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.

Support vector machines

Support vector machine

26.1 Giới thiệu

Support vector machine (SVM) là một trong những thuật toán phân lớp phổ biến và hiệu quả. Ý tưởng đứng sau SVM khá đơn giản, nhưng để hiểu được cách tìm nghiệm của nó, chúng ta cần một chút kiến thức về tối ưu và *duality*.

Trước khi đi vào phần ý tưởng chính của SVM, chúng ta cùng ôn lại kiến thức về hình học giải tích trong chương trình phổ thông.

26.1.1 Khoảng cách từ một điểm tới một siêu mặt phẳng

Trong không gian hai chiều, khoảng cách từ một điểm có tọa độ (x_0, y_0) tới *đường thẳng* có phương trình $w_1x + w_2y + b = 0$ được xác định bởi

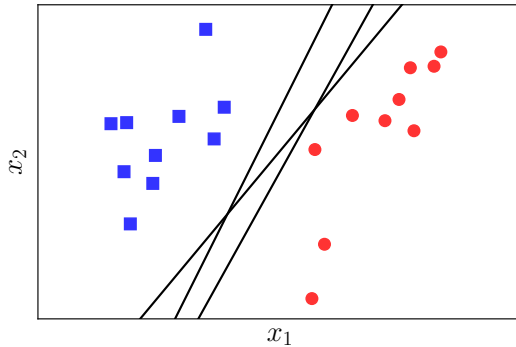
$$\frac{|w_1x_0 + w_2y_0 + b|}{\sqrt{w_1^2 + w_2^2}}$$

Trong không gian ba chiều, khoảng cách từ một điểm có tọa 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 bỏ dấu trị tuyệt đối ở tử số, 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 một phía (tạm gọi là *phía dương*), những điểm làm cho giá trị này mang dấu âm nằm về phía còn lại (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.

Các công thức này có thể được tổng quát lên cho trường hợp không gian d chiều. Khoảng cách từ một điểm (vector) có tọa độ $(x_{10}, x_{20}, \dots, x_{d0})$ tới *siêu mặt phẳng* (*hyperplane*) có



Hình 26.1: Hai lớp dữ liệu đỏ và xanh là *linearly separable*. Có vô số các đường thẳng có thể phân tách chính xác hai lớp dữ liệu này (Xem thêm Chương 13–Perceptron learning algorithm).

phương trình $w_1x_1 + w_2x_2 + \dots + w_dx_d + b = 0$ được xác định bởi

$$\frac{|w_1x_{10} + w_2x_{20} + \dots + w_dx_{d0} + b|}{\sqrt{w_1^2 + w_2^2 + \dots + w_d^2}} = \frac{|\mathbf{w}^T \mathbf{x}_0 + b|}{\|\mathbf{w}\|_2}$$

với $\mathbf{x}_0 = [x_{10}, x_{20}, \dots, x_{d0}]^T$, $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$.

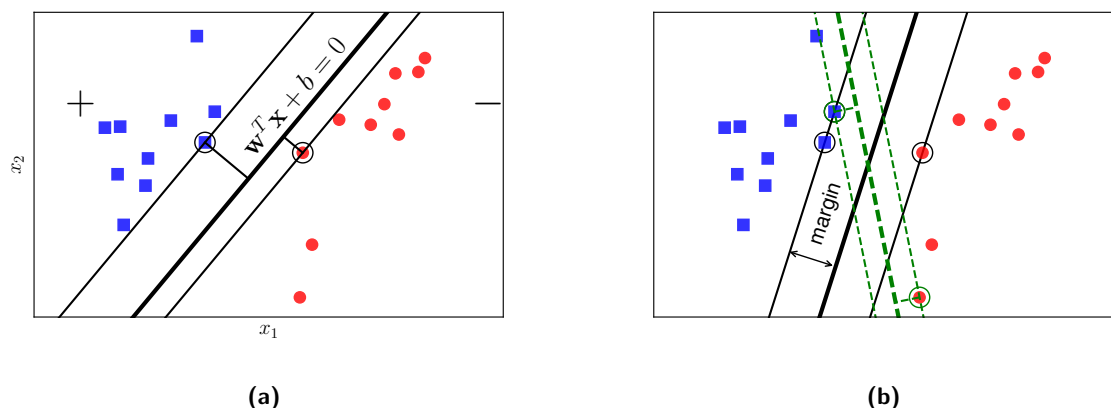
26.1.2 Nhắc lại bài toán phân chia hai lớp dữ liệu

Chúng ta cùng quay lại với bài toán phân lớp như đã đề cập trong Chương 13 – Perceptron Learning Algorithm (PLA). Giả sử rằng có hai lớp dữ liệu được mô tả bởi các điểm (feature vector) trong không gian nhiều chiều, hơn nữa, hai lớp dữ liệu này là *linearly separable*, tức tồn tại một siêu phẳng phân chia chính xác hai lớp đó. Hãy tìm một siêu phẳng phân chia hai lớp đó đó, tức tất cả các điểm thuộc một lớp nằm về cùng một phía của siêu phẳng đó và ngược phía với toàn bộ các điểm thuộc lớp 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 26.1.

Có một câu hỏi được đặt ra ở đây. Trong vô số các mặt phân chia đó, đâu là mặt tốt nhất. Trong ba đường thẳng minh họa trong Hình 26.1, có hai đường thẳng khá *lệch* về phía lớp màu đỏ. Điều này có thể khiến cho lớp màu đỏ *không vui vì lãnh thổ bị lấn nhiều quá*. Việc này có thể khiến cho các điểm màu đỏ trong tương lai bị phân lớp lỗi thành điểm màu xanh. Liệu có cách nào để tìm được đường phân chia mà cả hai lớp đều cảm thấy *công bằng* và *hạnh phúc* nhất hay không?

Để trả lời câu hỏi này, chúng ta cần tìm một tiêu chuẩn để đo sự *hạnh phúc* của mỗi lớp. Nếu ta định nghĩa *mức độ hạnh phúc* của một lớp tỉ lệ thuận với *khoảng cách gần nhất* từ một điểm của lớp đó tới đường/mặt phân chia, ở Hình 26.2a, lớp màu đỏ sẽ *không được hạnh phúc cho lắm* vì đường phân chia gần nó hơn lớp màu xanh rất nhiều. Chúng ta cần một đường phân chia sao cho khoảng cách từ điểm gần nhất của mỗi lớp (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 *biên* hoặc *lề* (*margin*).

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



Hình 26.2: Ý tưởng của SVM. *Margin* của một lớp được định nghĩa là khoảng cách từ các điểm gần nhất của lớp đó tới mặt phân chia. *Margin* của hai lớp phải bằng nhau và lớn nhất có thể.

Xét tiếp Hình 26.2b khi khoảng cách từ đường phân chia tới các điểm gần nhất của mỗi lớp 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 lớp *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 lớp là rạch ròi hơn*. Bài toán tối ưu trong SVM chính là bài toán đi tìm đường phân chia sao cho *margin* giữa hai lớp 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.

26.2 Xây dựng bài toán tối ưu cho SVM

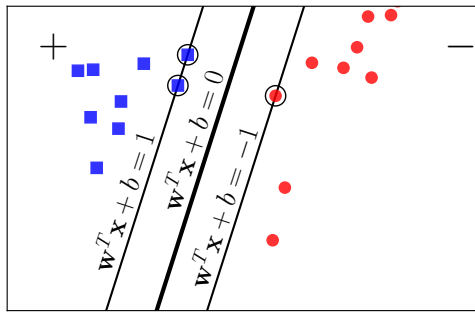
Giả sử rằng các cặp dữ liệu trong tập huấn luyện là $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ với vector $\mathbf{x}_i \in \mathbb{R}^d$ thể hiện *đầu vào* của một điểm dữ liệu và y_i là *nhãn* của điểm dữ liệu đó, d là số chiều của dữ liệu và N là số điểm dữ liệu. Giả sử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

Để dễ hình dung, chúng ta cùng làm với các ví dụ trong không gian hai chiều. Giả sử rằng các điểm màu xanh có nhãn là 1, các điểm tròn đỏ có nhãn là -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 lớp (Hình 26.3). Hơn nữa, lớp màu xanh nằm về *phía dương*, lớp màu đỏ nằm về *phía âm* của mặt phân chia. Nếu ngược lại, ta chỉ cần đổi dấu của \mathbf{w} và b . Ta cần đi tìm siêu phẳng được mô tả bởi các hệ số \mathbf{w} và b .

Ta quan sát thấy một điểm quan trọng như sau. 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 được 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 $(\mathbf{w}^T \mathbf{x}_n + b)$, vì vậy tử số luôn là một đại lượng không âm. Với mặt



Hình 26.3: Giả sử mặt phân chia có phương trình $\mathbf{w}^T \mathbf{x} + b = 0$. Không mất tính tổng quát, bằng cách nhân các hệ số \mathbf{w} và b với các hằng số phù hợp, ta có thể giả sử rằng điểm gần nhất của lớp màu xanh tới mặt này thoả mãn $\mathbf{w}^T \mathbf{x} + b = 1$. Khi đó, điểm gần nhất của lớp đỏ thoả mãn $\mathbf{w}^T \mathbf{x} + b = -1$.

phân chia này, *margin* được tính là khoảng cách gần nhất từ một điểm (trong cả hai lớp, vì cuối cùng *margin* của cả hai lớp sẽ như nhau) tới mặt đó, tức là

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

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

$$(\mathbf{w}, b) = \arg \max_{\mathbf{w}, b} \left\{ \min_n \frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2} \right\} = \arg \max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|_2} \min_n y_n(\mathbf{w}^T \mathbf{x}_n + b) \right\} \quad (26.1)$$

Có một nhận xét quan trọng 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 *bất kỳ* thì mặt phân chia không thay đổi, tức khoảng cách từ từng điểm đến mặt phân chia không đổi, tức *margin* không đổi. Vì vậy, ta có thể giả sử

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

với những điểm nằm gần mặt phân chia nhất (được khoanh tròn trong Hình 26.3).

Như vậy, với mọi n ta luôn có

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

Vậy bài toán tối ưu (26.1) có thể đưa về bài toán tối ưu có ràng buộc có dạng

$$(\mathbf{w}, b) = \arg \max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|_2} \quad (26.2)$$

$$\text{thoả mãn: } y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \forall n = 1, 2, \dots, N$$

Bằng một biến đổi đơn giản, ta có thể đưa bài toán này về dạng

$$(\mathbf{w}, b) = \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \quad (26.3)$$

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

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.

Có một quan sát rất quan trọng. Trong bài toán (26.3), 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 (26.3) 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 vì hàm mục tiêu là một *quadratic form*. 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. Thay vào đó, 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 một cách 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 nhất thiết *linearly separable*, như chúng ta sẽ thấy ở các chương sau của phần này.

Xác định lớp cho một điểm dữ liệu mới

Sau khi đã tìm được mặt phân cách $\mathbf{w}^T \mathbf{x} + b = 0$, nhãn của bất kỳ một điểm nào sẽ được xác định đơn giản bằng

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

26.3 Bài toán đối ngẫu của SVM

Nhắc lại rằng bài toán tối ưu (26.3) là một bài toán lồi. Chúng ta biết rằng nếu một bài toán lồi thỏa mãn tiêu chuẩn Slater thì *strong duality* thỏa mãn (xem Mục 25.3.2. Và nếu *strong duality* thỏa mãn thì nghiệm của bài toán chính là nghiệm của hệ điều kiện KKT (xem Mục 25.4.2).

26.3.1 Kiểm tra tiêu chuẩn Slater

Trong bước này, chúng ta sẽ chứng minh bài toán tối ưu (26.3) thỏa mãn điều kiện Slater. Điều kiện Slater nói rằng, nếu tồn tại \mathbf{w}, b thỏa mãn:

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

thì *strong duality* thỏa mãn. Việc kiểm tra này không quá phức tạp. Vì ta biết rằng luôn luôn có một siêu phẳng phân chia hai lớp nếu hai lớp đó là *linearly separable*, tức bài toán có nghiệm, nên *feasible set* của bài toán tối ưu (26.3) phải khác rỗng. Tức luôn luôn tồn tại cặp (\mathbf{w}_0, b_0) sao cho

$$1 - y_n(\mathbf{w}_0^T \mathbf{x}_n + b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (26.4)$$

$$\Leftrightarrow 2 - y_n(2\mathbf{w}_0^T \mathbf{x}_n + 2b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (26.5)$$

Vậy ta 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 thỏa mãn.

26.3.2 Lagrangian của bài toán SVM

Lagrangian của bài toán (26.3) là

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \frac{1}{2} \|\mathbf{w}\|_2^2 + \sum_{n=1}^N \lambda_n (1 - y_n (\mathbf{w}^T \mathbf{x}_n + b)) \quad (26.6)$$

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

26.3.3 Hàm đối ngẫu Lagrange

Theo định nghĩa, hàm đối ngẫu Lagrange 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:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \mathbf{w} - \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n = \mathbf{0} \Rightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (26.7)$$

$$\nabla_b \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda}) = \sum_{n=1}^N \lambda_n y_n = 0 \quad (26.8)$$

Thay (26.7) và (26.8) vào (26.6) ta thu được $g(\boldsymbol{\lambda})$ ¹:

$$g(\boldsymbol{\lambda}) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \quad (26.9)$$

Hàm $g(\boldsymbol{\lambda})$ trong (26.9) là **hàm số quan trọng nhất trong SVM**, chúng ta sẽ thấy rõ hơn trong chương Kernel SVM.

Bằng cách ký hiệu 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 $g(\boldsymbol{\lambda})$ dưới dạng²

$$g(\boldsymbol{\lambda}) = -\frac{1}{2} \boldsymbol{\lambda}^T \mathbf{V}^T \mathbf{V} \boldsymbol{\lambda} + \mathbf{1}^T \boldsymbol{\lambda}. \quad (26.10)$$

Nếu đặt $\mathbf{K} = \mathbf{V}^T \mathbf{V}$ thì \mathbf{K} là một ma trận nửa xác định dương. Thật vậy, với mọi vector $\boldsymbol{\lambda}$, ta có $\boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} = \boldsymbol{\lambda}^T \mathbf{V}^T \mathbf{V} \boldsymbol{\lambda} = \|\mathbf{V} \boldsymbol{\lambda}\|_2^2 \geq 0$. Vậy $g(\boldsymbol{\lambda}) = -\frac{1}{2} \boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} + \mathbf{1}^T \boldsymbol{\lambda}$ là một hàm *concave*.

¹ Phần chứng minh coi như một bài tập nhỏ cho bạn đọc.

² Phần chứng minh coi như một bài tập nhỏ khác cho bạn đọc.

26.3.4 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 của bài toán (26.3) có dạng

$$\begin{aligned} \boldsymbol{\lambda} &= \arg \max_{\boldsymbol{\lambda}} g(\boldsymbol{\lambda}) \\ \text{thỏa mãn: } \boldsymbol{\lambda} &\succeq 0 \\ \sum_{n=1}^N \lambda_n y_n &= 0 \end{aligned} \quad (26.11)$$

Ràng buộc thứ hai được lấy từ (26.8). 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*³. Hơn nữa, bài toán này là một quadratic programming và cũng có thể được giải bằng các thư viện như CVXOPT.

Trong bài toán đối ngẫu này, số lượng tham số phải tìm là N , là chiều của $\boldsymbol{\lambda}$, cũng chính là số điểm dữ liệu. Trong khi đó, với bài toán gốc (26.3), 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 tập huấn luyện 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 cấu trúc đặc biệt của hệ điều kiện KKT. Ngoài ra, dạng đặc biệt của bài toán đối ngẫu giúp các nhà khoa học đã phát triển thêm một dạng tổng quát của SVM, khiến nó hoạt động cả với trường hợp dữ liệu hai lớp là không *linear separable*. Chúng ta sẽ bàn kỹ tới trường hợp này trong chương Kernel SVM.

26.3.5 Điều kiện KKT

Quay trở lại bài toán, vì đây là một bài toán lồi và *strong duality* thỏa mãn, nghiệm của bài toán sẽ thỏa mãn hệ điều kiện KKT sau đây với biến số là \mathbf{w}, b và $\boldsymbol{\lambda}$.

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (26.12)$$

$$\lambda_n \geq 0, \quad \forall n = 1, 2, \dots, N \quad (26.13)$$

$$\lambda_n(1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0, \quad \forall n = 1, 2, \dots, N \quad (26.14)$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (26.15)$$

$$\sum_{n=1}^N \lambda_n y_n = 0 \quad (26.16)$$

Trong những điều kiện trên, điều kiện (26.14) 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 $\mathbf{w}^T \mathbf{x}_n + b = y_n$, với chú ý rằng $y_n^2 = 1, \forall n$.

³ Không chỉ riêng với bài toán tối ưu của SVM, các bài toán đối ngẫu luôn là bài toán lồi. Ở đây chúng ta chỉ khẳng định lại tính chất đó.

Những điểm thoả mãn (26.3.5) chính là những điểm nằm gần mặt phân chia nhất, là những điểm được khoanh tròn trong Hình 26.3. Hai đường thẳng $\mathbf{w}^T \mathbf{x}_n + b = \pm 1$ tựa lên các điểm thoả mãn (26.3.5). Những điểm (vector) thoả mãn (26.3.5) còn được gọi là các *support vector*. Và từ đó, cái tên *support vector machine* ra đời.

Một quan sát khác, số lượng những điểm thoả mãn (26.3.5) thường chiếm số lượng rất nhỏ trong số N điểm của tập huấn luyện. Chỉ cần dựa trên những *support vector* này, chúng ta hoàn toàn có thể xác định được mặt phân cách cần tìm. Nói cách khác, hầu hết các λ_n bằng 0, tức 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 hơn các mô hình tương tự với nghiệm là *dense* (hầu hết các phần tử 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. Phương pháp thường được dùng để giải hệ này là *sequential minimal optimization* (SMO) [Pla98, ZYX+08]. Trong phương pháp này, các cặp hai nhân tử Lagrange (hai thành phần của λ) được chọn ra để tối ưu tại mỗi vòng lặp. Trong các bài báo trên, việc chọn cặp như thế nào được nêu rõ. Việc này được thực hiện nhiều lần cho tới khi thuật toán hội tụ [Bis06].

Chúng ta 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 (26.11) bằng CVXOPT với một ví dụ nhỏ và bằng thư viện `sklearn` (có thể áp dụng cho trường hợp nhiều điểm dữ liệu và nhiều chiều dữ liệu hơn).

Sau khi tìm được λ từ bài toán (26.11), ta có thể suy ra được \mathbf{w} dựa vào (26.15) và b dựa vào (26.14) và (26.16). Rõ ràng ta chỉ cần quan tâm tới $\lambda_n \neq 0$.

Đặt $\mathcal{S} = \{n : \lambda_n \neq 0\}$ và $N_{\mathcal{S}}$ là số phần tử của tập \mathcal{S} . Theo (26.15), \mathbf{w} được tính bằng

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (26.17)$$

Với mỗi $n \in \mathcal{S}$, ta có

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

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 trung bình cộng⁴ của tất cả các b tính được theo mỗi $n \in \mathcal{S}$

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (26.18)$$

⁴ Việc này cũng giống như cách làm trong các thí nghiệm vật lý. Để đo một đại lượng, người ta thường thực hiện việc đo nhiều lần rồi lấy kết quả trung bình để tránh sai số. Ở đây, về mặt toán học, b phải như nhau theo mọi cách tính; tuy nhiên, khi tính toán bằng máy tính, chúng ta có thể gặp các sai số nhỏ. Việc lấy trung bình sẽ làm giảm sai số đó.

Để xác định một điểm \mathbf{x} mới thuộc vào lớp nào, ta cần xác định dấu của biểu thức

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

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

26.4 Lập trình tìm nghiệm cho SVM

Trong mục này, chúng ta cùng tìm nghiệm cho SVM bằng hai cách khác nhau. Cách thứ nhất dựa theo bài toán (26.11) và các công thức (26.18) và (26.17). Cách thứ hai sử dụng trực tiếp thư viện `sklearn`. Cách thứ nhất giúp chứng minh tính đúng đắn của các công thức đã xây dựng. Cách thứ hai sẽ giúp các bạn biết cách áp dụng SVM vào dữ liệu thực tế.

26.4.1 Tìm nghiệm theo công thức

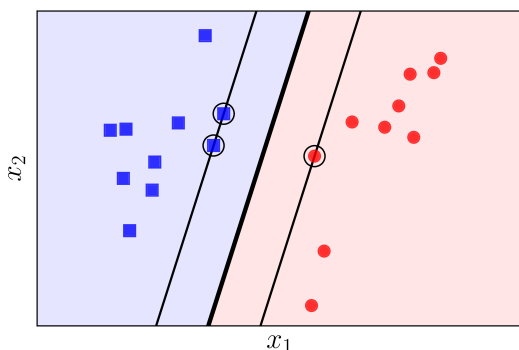
Trước tiên chúng ta gọi các thư viện cần dùng và tạo dữ liệu giả (dữ liệu này được sử dụng trong các hình vẽ từ đầu chương. Ta thấy rằng hai class là *linearly separable*).

```

from __future__ import print_function
import numpy as np
np.random.seed(22)
# simulated samples
means = [[2, 2], [4, 2]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N) # blue class data
X1 = np.random.multivariate_normal(means[1], cov, N) # red class data
X = np.concatenate((X0, X1), axis = 0) # all data
y = np.concatenate((np.ones(N), -np.ones(N)), axis = 0) # label
# solving the dual problem (variable: lambda)
from cvxopt import matrix, solvers
V = np.concatenate((X0, -X1), axis = 0) # V in the book
Q = matrix(V.dot(V.T))
p = matrix(-np.ones((2*N, 1))) # objective function 1/2 lambda^T*Q*lambda - 1^T*lambda
# build A, b, G, h
G = matrix(-np.eye(2*N))
h = matrix(np.zeros((2*N, 1)))
A = matrix(y.reshape(1, -1))
b = matrix(np.zeros((1, 1)))
solvers.options['show_progress'] = False
sol = solvers.qp(Q, p, G, h, A, b)
l = np.array(sol['x']) # solution lambda

# calculate w and b
w = Xbar.T.dot(l)
S = np.where(l > 1e-8)[0] # support set, 1e-8 to avoid small value of l.
b = np.mean(y[S].reshape(-1, 1) - X[S,:].dot(w))
print('Number of suport vectors = ', S.size)
print('w = ', w.T)
print('b = ', b)

```

Hình 26.4: 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 lam. Điều tương tự xảy ra với các điểm nằm trên nền màu đỏ nhạt.

Kết quả:

```
Number of support vectors = 3
w = [[-2.00984382  0.64068336]]
b = 4.66856068329
```

Như vậy trong số 20 điểm dữ liệu của cả hai lớp, chỉ có ba điểm nằm trong *support set*, tức có ba điểm đóng vai trò là các *support vector*. Ba điểm này giúp xây dựng đường thẳng phân chia với w và b như đã tính được. Kết quả tìm được được minh họa trong Hình 26.4. Đường màu đen đậm ở giữa chính là mặt phân cách tìm được bằng SVM. Các đường đen mảnh thể hiện các đường thẳng *tựa* lên các *support vector* được khoanh tròn.

Các hình vẽ và source code trong bài có thể được tìm thấy tại <https://goo.gl/VKBgVG>.

26.4.2 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:

```
# solution by sklearn
from sklearn.svm import SVC

model = SVC(kernel = 'linear', C = 1e5) # just a big number
model.fit(X, y)

w = model.coef_
b = model.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, với cách làm đơn giản hơn rất nhiều. Có rất nhiều tùy chọn cho SVM, trong đó có thuộc tính **kernel**, các bạn sẽ dần thấy trong các chương sau.

26.5 Tóm tắt và thảo luận

- Với bài toán phân lớp nhị phân mà hai lớp dữ liệu là *linearly separable*, có vô số các mặt phân cách phẳng giúp phân chia hai lớp đó. Khoảng cách gần nhất từ một điểm dữ liệu tới mặt phân cách ấy được gọi là *margin* của bộ phân lớp với ranh giới là mặt phẳng đó.
- Support vector machine là bài toán đi tìm mặt phân cách sao cho *margin* có được là lớn nhất, đồng nghĩa với việc các điểm dữ liệu có một *khoảng cách an toàn* tới mặt phân cách.
- Bài toán tối ưu trong SVM là một bài toán *convex* với hàm mục tiêu là *strictly convex*, vì vậy, *local optimum* cũng là *global optimum* của bài toán. Hơn nữa, bài toán tối ưu đó là một *quadratic programming* (QP).
- Mặc dù có thể trực tiếp giải SVM qua bài toán *primal*, thông thường người ta thường giải bài toán *dual*. Bài toán *dual* cũng là một QP nhưng nghiệm là *sparse* nên có những phương pháp giải hiệu quả hơn.

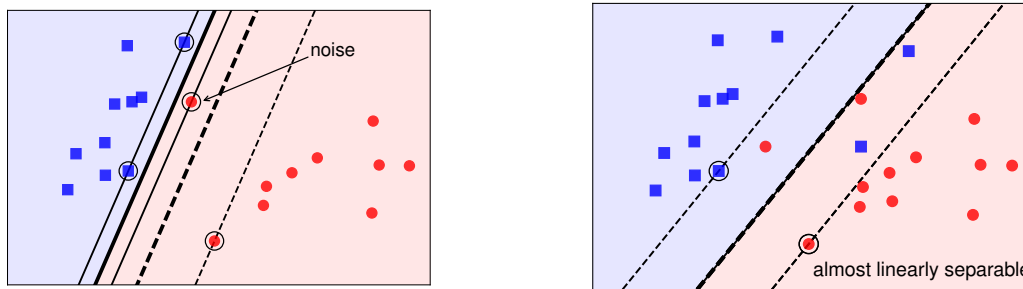
Soft-margin support vector machine

27.1 Giới thiệu

Giống như perceptron learning algorithm (PLA), support vector machine (SVM) *thuần* chỉ làm việc khi dữ liệu của hai lớp 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.

Xét hai ví dụ trong Hình 27.1. 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 27.1a nhưng có một điểm *nhều* của lớp đỏ ở quá gần so với lớp 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 với các điểm ở lớp xanh và quá xa các điểm thuộc lớp 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 27.1b. Trong trường hợp này, không tồn tại đường thẳng nào hoàn toàn phân chia hai lớp dữ liệu, vì vậy bài toán tối ưu SVM trở nên vô nghiệm. Tuy nhiên, nếu *chịu hy sinh một chút* những điểm ở gần khu vực biên giới giữa hai lớp, 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 *lấn* sang phía bên kia của các *đường support* tương ứng, ta gọi điểm đó rơi vào *vùng không an toàn*. Như trong hình, hai điểm màu đỏ nằm phía bên trái *đường support* của lớp đỏ được xếp vào loại không an toàn, mặc dù có một điểm đỏ vẫn nằm trong khu vực nền màu đỏ. Hai điểm màu xanh ở phía phải của *đường support* của lớp xanh thậm chí đều lấn sang phần có nền màu đỏ.



(a) Khi có nhiễu nhỏ.

(b) Khi dữ liệu gần linearly separable.

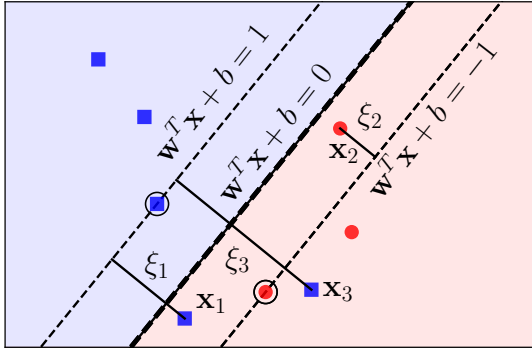
Hình 27.1: Hai trường hợp khi support vector machine *thuần* làm việc không hiệu quả. (a) Hai lớp vẫn linearly separable nhưng một điểm thuộc class này quá gần class kia, điểm này có thể là nhiễu. (b) Dữ liệu hai lớp không linearly separable, mặc dù chúng đã gần linearly separable.

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*). Để phân biệt, SVM *thuần* còn được gọi là *hard-margin SVM* (*SVM biên cứng*). SVM chấp nhận một vài điểm trong tập huấn luyện bị phân lớp lỗi này được gọi là *soft-margin SVM*.

Có hai cách xây dựng và giải quyết bài toán tối ưu cho *soft-margin SVM*, 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 như sẽ được thấy ở các chương sau của cuốn sách này. 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 sẽ được đề cập trong chương tiếp theo. 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* sẽ được đề cập trong chương (*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. Tiếp theo, chúng ta cùng đi phân tích bài toán tối ưu cho *soft-margin SVM*.

27.2 Phân tích toán học

Như đã đề cập phía trên, để có một *margin* lớn hơn trong *soft margin SVM*, ta cần *hy sinh* một vài điểm dữ liệu bằng cách chấp nhận cho chúng rơi vào vùng *không an toàn*. Tất nhiên, việc *hy sinh* này cần được hạn chế, nếu không, 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* cũng như tối thiểu *sự hy sinh*.



Hình 27.2: Giới thiệu các biến slack ξ_n . Với các điểm nằm ở *khu vực an toàn*, $\xi_n = 0$. Những điểm nằm trong vùng không an toàn, nhưng vẫn đúng phía so với đường ranh giới (đường nét đứt đậm), 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 .

Giống như với *hard-margin SVM*, việc tối đa *margin* có thể đưa về việc tối thiểu $\|\mathbf{w}\|_2^2$. Để đong đếm *sự hy sinh*, chúng ta cùng theo dõi Hình 27.2. 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* (nằm đúng vào màu nền tương ứng và nằm ngoài *khu vực margin*), $\xi_n = 0$, tức không có *sự hy sinh mất mát* nào xảy ra. Với mỗi điểm nằm trong *vùng không an toàn* như $\mathbf{x}_1, \mathbf{x}_2$ hay \mathbf{x}_3 , ta cần có $\xi_i > 0$, tức *mất mát* đã xảy ra. Đại lượng này nên tỉ lệ với khoảng cách từ điểm vi phạm tương ứng tới biên giới an toàn. Nhận thấy rằng nếu $y_i = \pm 1$ là *nhãn* của \mathbf{x}_i trong *vùng không an toàn* thì ξ_i có thể được định nghĩa là

$$\xi_i = |\mathbf{w}^T \mathbf{x}_i + b - y_i| \quad (27.1)$$

(ở đây, ta đã bỏ mẫu số $\|\mathbf{w}\|_2$ đi vì ta chỉ cần một đại lượng tỉ lệ thuận.) Nhắc lại bài toán tối ưu cho *hard-margin SVM*:

$$(\mathbf{w}, b) = \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \quad (27.2)$$

$$\text{thoả mãn: } y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \quad \forall n = 1, 2, \dots, N$$

Với *soft-margin SVM*, hàm mục tiêu sẽ có thêm một số hạng nữa giúp tối thiểu *tổng sự hy sinh*. Từ đó ta có hàm mục tiêu

$$\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \quad (27.3)$$

trong đó C là một hằng số dương. Điều kiện ràng buộc được thay đổi một chút. Với mỗi cặp dữ liệu (\mathbf{x}_n, y_n) , thay vì ràng buộc *cứng* $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$, ta sử dụng 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 $\xi_n \geq 0, \quad \forall n = 1, 2, \dots, N$.

Tóm lại, ta sẽ có bài toán tối ưu *primal* cho *soft-margin SVM* như sau đây.

$$(\mathbf{w}, b, \xi) = \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \quad (27.4)$$

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

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 vì vùng an toàn bị nhỏ đi. 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 lớp 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 (27.4) có thêm sự xuất hiện của các biến *slack* ξ_n . Các $\xi_n = 0$ tương ứng với những điểm dữ liệu nằm trong *vùng an toàn*. Các $0 < \xi_n \leq 1$ 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. Các $\xi_n > 1$ tương ứng với các điểm bị phân lớp sai.
- Hàm mục tiêu trong bài toán tối ưu (27.4) 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 (27.4) 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 (27.4) bằng hai cách khác nhau.

27.3 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 của nó.

Trước kết, ta cần kiểm tra tiêu chuẩn Slater cho bài toán tối ưu lồi (27.4). Nếu tiêu chuẩn này được thoả mãn, *strong duality* sẽ thoả mãn, và ta sẽ có nghiệm của bài toán tối ưu (27.4) là nghiệm của hệ điều kiện KKT (xem Chương 25).

27.3.1 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 $y_n(\mathbf{w}^T \mathbf{x}_n + b) + \xi_n > 1, \forall n = 1, 2, \dots, N$. Vì vậy, bài toán này thoả mãn tiêu chuẩn Slater.

27.3.2 Lagrangian của bài toán Soft-margin SVM

Lagrangian cho bài toán (27.4) là

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\xi}, \boldsymbol{\lambda}, \boldsymbol{\mu}) = \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \lambda_n (1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) - \sum_{n=1}^N \mu_n \xi_n \quad (27.5)$$

với $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_N]^T \succeq 0$ và $\boldsymbol{\mu} = [\mu_1, \mu_2, \dots, \mu_N]^T \succeq 0$ là các biến đối ngẫu Lagrange.

27.3.3 Bài toán đối ngẫu

Hàm số đối ngẫu của bài toán tối ưu (27.4) là:

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

Với mỗi cặp $(\boldsymbol{\lambda}, \boldsymbol{\mu})$, chúng ta sẽ quan tâm tới $(\mathbf{w}, b, \boldsymbol{\xi})$ thoả mãn điều kiện đạo hàm của Lagrangian bằng 0:

$$\nabla_{\mathbf{w}} \mathcal{L} = 0 \Leftrightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (27.6)$$

$$\nabla_b \mathcal{L} = 0 \Leftrightarrow \sum_{n=1}^N \lambda_n y_n = 0 \quad (27.7)$$

$$\nabla_{\xi_n} \mathcal{L} = 0 \Leftrightarrow \lambda_n = C - \mu_n \quad (27.8)$$

Từ (27.8) ta thấy rằng ta chỉ quan tâm tới những cặp $(\boldsymbol{\lambda}, \boldsymbol{\mu})$ 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 biểu thức Lagrangian (27.5), ta thu được hàm mục tiêu của bài toán đối ngẫu¹

$$g(\boldsymbol{\lambda}, \boldsymbol{\mu}) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \quad (27.9)$$

Chú ý rằng hàm này không phụ thuộc vào $\boldsymbol{\mu}$ nhưng ta cần lưu ý ràng buộc (27.8), ràng buộc này và điều kiện không âm của $\boldsymbol{\lambda}$ có thể được viết gọn lại thành $0 \leq \lambda_n \leq C$, khi đó ta đã giảm được biến $\boldsymbol{\mu}$. Lúc này, bài toán đối ngẫu trở thành

$$\boldsymbol{\lambda} = \arg \max_{\boldsymbol{\lambda}} g(\boldsymbol{\lambda})$$

$$\text{thoả mãn: } \sum_{n=1}^N \lambda_n y_n = 0 \quad (27.10)$$

$$0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \quad (27.11)$$

Bài toán này gần giống với bài toán đối ngẫu của *hard-margin SVM*, chỉ khác là có thêm ràng buộc mỗi λ_n bị chặn trên bởi C . Khi C rất lớn, ta có thể coi hai bài toán là như nhau.

¹ Bạn đọc hãy coi đây như là một bài tập nhỏ.

Ràng buộc (27.11) còn được gọi là *box constraint* vì không gian các điểm $\boldsymbol{\lambda}$ thoả 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 $(\mathbf{w}, b, \boldsymbol{\xi})$ của bài toán gốc. Trước hết, chúng ta cùng xem xét hệ điều kiện KKT và các tính chất của nghiệm.

27.3.4 Hệ điều kiện KKT

(Bạn đọc không muốn đi sâu vào toán có thể bỏ qua mục này)

Hệ điều kiện KKT² của bài toán tối ưu *soft-margin SVM* là, với mọi $n = 1, 2, \dots, N$:

$$1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0 \tag{27.12}$$

$$-\xi_n \leq 0 \tag{27.13}$$

$$\lambda_n \geq 0 \tag{27.14}$$

$$\mu_n \geq 0 \tag{27.15}$$

$$\lambda_n(1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0 \tag{27.16}$$

$$\mu_n \xi_n = 0 \tag{27.17}$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \tag{27.6}$$

$$\sum_{n=1}^N \lambda_n y_n = 0 \tag{27.7}$$

$$\lambda_n = C - \mu_n \tag{27.8}$$

Từ (27.8) ta thấy chỉ có những n ứng với $\lambda_n > 0$ mới đóng góp vào nghiệm \mathbf{w} của bài toán. Tập hợp $\mathcal{S} = \{n : \lambda_n > 0\}$ được gọi là *support set*, và $\{\mathbf{x}_n, n \in \mathcal{S}\}$ được gọi là tập các điểm *support vectors*.

Khi $\lambda_n > 0$, (27.16) chỉ ra rằng

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1 - \xi_n \tag{27.18}$$

Nếu có thêm điều kiện $0 < \lambda_n < C$, (27.11) nói rằng $\mu_n = C - \lambda_n > 0$, kết hợp với (27.17), ta thu được $\xi_n = 0$. Tiếp tục kết hợp với (27.18), ta suy ra $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$. Nói cách khác,

$$\mathbf{w}^T \mathbf{x}_n + b = y_n, \quad \forall n : 0 < \lambda_n < C \tag{27.19}$$

Tóm lại, khi $0 < \lambda_n < C$, các điểm \mathbf{x}_n nằm *chính xác* trên các *margin* (hai đường nét đứt mảnh trong Hình 27.2). Tương tự như với *hard-margin SVM*, giá trị b có thể được tính theo công thức (*numerical stable solution*):

$$b = \frac{1}{N_{\mathcal{M}}} \sum_{m \in \mathcal{M}} (y_m - \mathbf{w}^T \mathbf{x}_m) \tag{27.20}$$

² Để cho dễ hình dung, các điều kiện (27.6) (27.7) (27.8) đã được nhắc lại trong hệ này.

với $\mathcal{M} = \{m : 0 < \lambda_m < C\}$ và $N_{\mathcal{M}}$ là số phần tử của \mathcal{S} . Nghiệm của bài toán *soft-margin SVM* được cho bởi (27.8) và (27.20).

Nghiệm của bài toán *soft-margin SVM*

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (27.21)$$

$$b = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (27.22)$$

Cũng từ (27.18) và (27.16) ta suy ra $y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 1$ với những điểm tương ứng với $\lambda_n = C$. Tức những điểm này nằm giữa hoặc trên hai đường *margin*. 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 đường *margin*.

Mục đích cuối cùng là xác định nhãn cho một điểm mới hơn là tính cụ thể \mathbf{w} và b . Vì vậy, ta quan tâm hơn tới cách xác định giá trị của biểu thức sau đây với \mathbf{x} bất kỳ:

$$\mathbf{w}^T \mathbf{x} + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (27.23)$$

Trong cách tính này, nếu biết cách tính các tích vô hướng $\mathbf{x}_m^T \mathbf{x}$ và $\mathbf{x}_m^T \mathbf{x}_n$, ta có thể xác định được bộ phân lớp. Trong chương tiếp theo, ta sẽ thấy rằng bằng cách sử dụng các *phép biến đổi phi tuyến (nonlinear transformation)* để thay đổi tích vô hướng bằng các hàm khác, ta sẽ thu được các bộ phân lớp làm việc hiệu quả với dữ liệu không *linear separable*.

27.4 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 (27.4) về một bài toán tối ưu không ràng buộc, và có khả năng giải được bằng các phương pháp gradient descent giống như các neural network. Đây cũng là nền tảng để kết hợp một *multi-class SVM* vào các neural network, như sẽ được trình bày trong Chương 29.

27.4.1 Bài toán tối ưu không ràng buộc tương đương

Để ý thấy rằng điều kiện ràng buộc thứ nhất:

$$1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x} + b) \leq 0 \Leftrightarrow \xi_n \geq 1 - y_n(\mathbf{w}^T \mathbf{x} + b) \quad (27.24)$$

Kết hợp với điều kiện $\xi_n \geq 0$ ta sẽ thu được bài toán ràng buộc tương đương với bài toán (27.4) như sau:

$$(\mathbf{w}, b, \xi) = \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \quad (27.25)$$

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

Tiếp theo, để đưa bài toán (27.25) 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 (27.25), tức tại đó hàm mục tiêu đạt giá trị nhỏ nhất, thì

$$\xi_n = \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)), \quad \forall n = 1, 2, \dots, N \quad (27.26)$$

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

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

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! Điều mâu thuẫn này chỉ ra rằng nhận xét (27.26) là chính xác.

Khi đó, bằng cách thay toàn bộ các giá trị của ξ_n trong (27.26) vào hàm mục tiêu, ta thu được bài toán tối ưu

$$(\mathbf{w}, b, \xi) = \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \quad (27.27)$$

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

Từ đây ta thấy rằng biến số ξ không còn quan trọng trong bài toán này nữa, ta có thể lược bỏ ràng buộc này mà không làm thay đổi nghiệm của bài toán.

Bài toán (27.27) tương đương với

$$(\mathbf{w}, b) = \arg \min_{\mathbf{w}, b} \left\{ \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \triangleq J(\mathbf{w}, b) \right\} \quad (27.28)$$

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. Góc nhìn mới này giúp xây dựng hàm mất mát $J(\mathbf{w}, b)$ một cách *tự nhiên* hơn bằng cách sử dụng một hàm số có tên là *hinge loss*.

27.4.2 Hinge loss

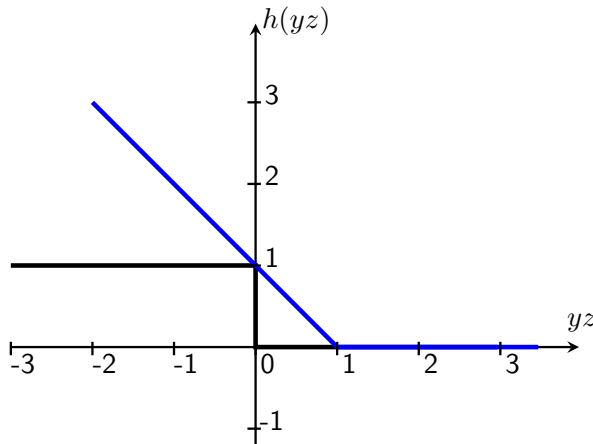
Nhắc lại một chút về hàm *cross entropy*. Với mỗi cặp hệ số (\mathbf{w}, b) và cặp dữ liệu (\mathbf{x}_n, y_n) , đặt $z_n = \mathbf{w}^T \mathbf{x}_n + b$ và $a_n = \sigma(z_n)$ (σ là *sigmoid function*). Hàm *cross entropy* được định nghĩa là

$$J_n^1(\mathbf{w}, b) = -(y_n \log(a_n) + (1 - y_n) \log(1 - a_n)) \quad (27.29)$$

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 bộ phân lớp. Hàm số này có dạng

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



Hình 27.3: Hinge loss (màu xanh) và zero-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 về phía trái gây ra mất mát nhiều hơn.

Hàm này có tên là *hinge loss*. Trong đó, $z_n = \mathbf{w}^T \mathbf{x}_n + b$ 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. Chúng ta sẽ sớm thấy ý nghĩa của hàm này thông qua đồ thị của hàm tương ứng. Hình 27.3 mô tả đồ thị hàm số *hinge loss*³ $f(yz) = \max(0, 1 - yz)$ và so sánh với hàm *zero-one loss*. Hàm *zero-one loss* là hàm *đếm số điểm bị phân lớp lỗi*. Trong Hình 27.5, biến số là yz là tích của đầu ra mong muốn (ground truth) và *score* z . Những điểm ở phía phải của trục tung ứng với những điểm được phân loại đúng, tức z tìm được cùng dấu với y . Những điểm ở phía trái của trục tung ứng với các điểm bị phân loại sai. Ta có các nhận xét sau đây:

- Với hàm *zero-one loss*, các điểm có *score* ngược dấu với đầu ra mong muốn ($yz < 0$) sẽ gây ra mất mát như nhau (bằng 1), bất kể chúng ở gần hay xa đường ranh giới (trục tung). Đây là một hàm rời rạc, rất khó tối ưu và ta cũng khó có thể đo 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 $yz \geq 1$, sẽ không gây ra mất mát gì. Những điểm nằm giữa margin của lớp tương ứng và đường ranh giới ứng với $0 < yz < 1$. Những điểm này gây ra một mất mát nhỏ (nhỏ hơn 1). Những điểm bị *misclassified*, tức $yz < 0$ sẽ gây ra mất mát lớn hơn. Vì vậy, khi tối thiểu hàm mất mát, ta sẽ hạn chế được những điểm bị *misclassified* và *lấn sang phần lãnh thổ* của lớp 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.

27.4.3 Xây dựng hàm mất mát

Xét bài toán *soft-margin SVM* bằng cách sử dụng *hinge loss*, với mỗi cặp (\mathbf{w}, b) , đặt

$$L_n(\mathbf{w}, b) = \max(0, 1 - y_n z_n) = \max(0, 1 - y_n (\mathbf{w}^T \mathbf{x}_n + b)) \quad (27.30)$$

³ Đồ thị của hàm số này có hình giống chiếc bản lề. Trong tiếng Anh, *hinge* nghĩa là bản lề.

Lấy trung bình cộng của các *loss* này theo mọi điểm dữ liệu trong tập huấn luyện ta được

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

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

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

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (27.31)$$

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

$$a - y_n(a\mathbf{w}^T \mathbf{x}_n + ab) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (27.32)$$

$$\Rightarrow 1 - y_n(a\mathbf{w}^T \mathbf{x}_n + ab) \leq 1 - a < 0, \quad \forall n = 1, 2, \dots, N \quad (27.33)$$

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* trở thành

$$J(\mathbf{w}, b) = \frac{1}{N} \left(\underbrace{\sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))}_{\text{hinge loss}} + \underbrace{\frac{\lambda}{2} \|\mathbf{w}\|_2^2}_{\text{regularization}} \right) \quad (27.34)$$

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 (27.34) giống với hàm mất mát (27.28) với $\lambda = \frac{1}{C}$, và thay việc lấy trung bình cộng bằng việc tính tổng.

Trong phần tiếp theo của mục này, chúng ta sẽ quan tâm tới bài toán tối ưu hàm mất mát được cho trong (27.34). Trước hết, đây là một hàm lồi theo \mathbf{w}, b vì các lý do sau:

- $1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)$ là một hàm tuyến tính theo \mathbf{w}, b nên nó là một hàm lồi. Hàm lấy giá trị lớn hơn trong hai hàm lồi là một hàm lồi, vì vậy, *hingle loss* là một hàm lồi.

- Hàm norm cũng là một hàm lồi.
- Tổng của hai hàm lồi là một hàm lồi.

Vì hàm mất mát là lồi, các thuật toán gradient descent với *learning rate* phù hợp sẽ giúp tìm được nghiệm của bài toán.

27.4.4 Tối ưu hàm mất mát

Vì việc tối ưu hàm mất mát dựa trên gradient descent, việc chính của mục này là tính đạo hàm của hàm mất mát theo \mathbf{w} và b .

Đạo hàm của phần *hinge loss* không quá phức tạp:

$$\nabla_{\mathbf{w}} (\max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))) = \begin{cases} -y_n \mathbf{x}_n & \text{nếu } 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 0 \\ \mathbf{0} & \text{o.w.} \end{cases} \quad (27.35)$$

$$\nabla_b (\max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))) = \begin{cases} -y_n & \text{nếu } 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 0 \\ 0 & \text{o.w.} \end{cases} \quad (27.36)$$

Phần *regularization* cũng có đạo hàm tương đối đơn giản:

$$\nabla_{\mathbf{w}} \left(\frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right) = \lambda \mathbf{w}; \quad \nabla_b \left(\frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right) = 0 \quad (27.37)$$

Nếu cập nhật bằng gradient descent thông qua chỉ một điểm dữ liệu (\mathbf{x}_n, y_n) (*stochastic gradient descent*). Nếu $1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) < 0$, ta không cập nhật gì và chuyển sang điểm tiếp theo. Ngược lại biểu thức cập nhật cho \mathbf{w}, b được cho bởi

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(-y_n \mathbf{x}_n + \lambda \mathbf{w}); \quad b \leftarrow b + \eta y_n \quad \text{nếu } 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 0 \quad (27.38)$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \lambda \mathbf{w}; \quad b \leftarrow b \quad \text{o.w.} \quad (27.39)$$

với η là *learning rate*. Với *mini-batch gradient descent* hoặc *batch gradient descent*, các biểu thức đạo hàm trên đây hoàn toàn có thể được lập trình bằng các kỹ thuật *vectorization*, như chúng ta sẽ thấy trong mục tiếp theo.

27.5 Lập trình với *soft-margin SVM*

Trong mục này, chúng ta 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 mất mát không ràng buộc bằng phương pháp gradient descent. Giá trị C được sử dụng là 100. Nếu mọi tính toán ở trên là chính xác, nghiệm của ba cách làm này sẽ gần giống nhau, khác nhau có thể một chút bởi sai số trong tính toán⁴. Chúng ta cũng sẽ thay C bởi những giá trị khác nhau và cùng xem các *margin* thay đổi như thế nào.

⁴ Ta có thể khẳng định việc này vì bài toán tối ưu *soft-margin SVM* là lồi.

Khai báo thư viện và tạo dữ liệu giả

```

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

means = [[2, 2], [4, 2]]
cov = [[.7, 0], [0, .7]]
N = 20 # number of samplers per class
X0 = np.random.multivariate_normal(means[0], cov, N) # each row is a data point
X1 = np.random.multivariate_normal(means[1], cov, N)
X = np.concatenate((X0, X1))
y = np.concatenate((np.ones(N), -np.ones(N)))

```

Các điểm màu xanh và đỏ trên Hình 27.4 minh họa các điểm dữ liệu của hai lớp. Dữ liệu này là *gần linearly separable*.

27.5.1 Giải bài toán bằng thư viện sklearn

```

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)

```

Kết quả:

```

w_sklearn = [[-1.87461946 -1.80697358]]
b_sklearn = 8.49691190196

```

27.5.2 Tìm nghiệm bằng cách giải bài toán đối ngẫu

Đoạn code dưới đây tương tự như việc giải bài toán *hard-margin SVM*, chỉ khác rằng ta có thêm ràng buộc về chặn trên của các nhân tử Lagrange:

```

from cvxopt import matrix, solvers
# build K
V = np.concatenate((X0, -X1), axis = 0) # V[n,:] = y[n]*X[n]
K = matrix(V.dot(V.T))
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 = np.vstack((np.zeros((2*N, 1)), C*np.ones((2*N, 1))))
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))) # continue on next page

```

```

solvers.options['show_progress'] = False
sol = solvers.qp(K, p, G, h, A, b)

l = np.array(sol['x']).reshape(2*N) # lambda vector

# support set
S = np.where(l > 1e-5)[0]
S2 = np.where(l < .999*C)[0]
# margin set
M = [val for val in S if val in S2] # intersection of two lists

VS = V[S] # shape (NS, d)
lS = l[S] # shape (NS,)
w_dual = lS.dot(VS) # shape (d,)
yM = y[M] # shape (NM,)
XM = X[M] # shape (NM, d)
b_dual = np.mean(yM - XM.dot(w_dual)) # shape (1,)
print('w_dual = ', w_dual)
print('b_dual = ', b_dual)

```

Kết quả:

```

w_dual = [-1.87457279 -1.80695039]
b_dual = 8.49672109814

```

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

27.5.3 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 $\text{lam} = 1/C$. Trước hết ta viết các hàm tính giá trị hàm mất mát và đạo hàm theo \mathbf{w} và b .

```

lam = 1./C
def loss(X, y, w, b):
    """
    X.shape = (2N, d), y.shape = (2N,), w.shape = (d,), b is a scalar
    """
    z = X.dot(w) + b # shape (2N,)
    yz = y*z
    return (np.sum(np.maximum(0, 1 - yz)) + .5*lam*w.dot(w))/X.shape[0]

def grad(X, y, w, b):
    z = X.dot(w) + b # shape (2N,)
    yz = y*z # element wise product, shape (2N,)
    active_set = np.where(yz <= 1)[0] # consider 1 - yz >= 0 only
    _yX = - X*y[:, np.newaxis] # each row is y_n*x_n
    grad_w = (np.sum(_yX[active_set], axis = 0) + lam*w)/X.shape[0]
    grad_b = (-np.sum(y[active_set]))/X.shape[0]
    return (grad_w, grad_b) ## continue on next page

```

```

def num_grad(X, y, w, b):
    eps = 1e-10
    gw = np.zeros_like(w)
    gb = 0
    for i in xrange(len(w)):
        wp = w.copy()
        wm = w.copy()
        wp[i] += eps
        wm[i] -= eps
        gw[i] = (loss(X, y, wp, b) - loss(X, y, wm, b))/(2*eps)
    gb = (loss(X, y, w, b + eps) - loss(X, y, w, b - eps))/(2*eps)
    return (gw, gb)

w = .1*np.random.randn(X.shape[1])
b = np.random.randn()
(gw0, gb0) = grad(X, y, w, b)
(gw1, gb1) = num_grad(X, y, w, b)
print('grad_w difference = ', np.linalg.norm(gw0 - gw1))
print('grad_b difference = ', np.linalg.norm(gb0 - gb1))

```

Kết quả:

```

grad_w difference = 1.27702840067e-06
grad_b difference = 4.13701854995e-08

```

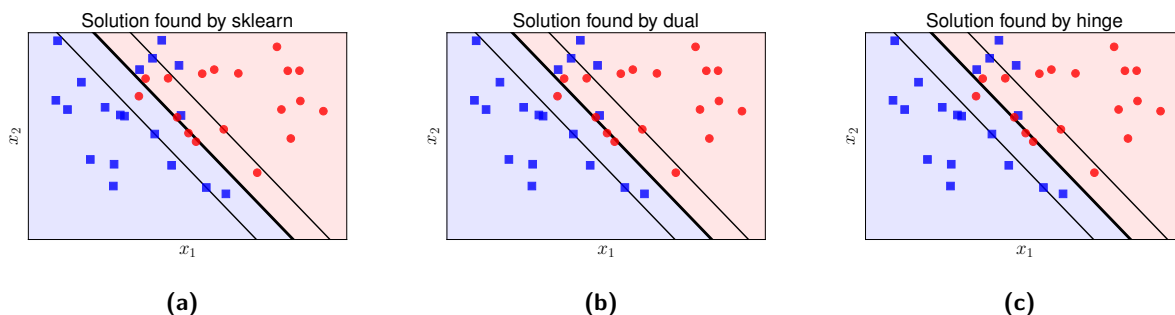
Sự sai khác giữa hai cách tính là nhỏ; vậy ta có thể tin tưởng sử dụng hàm `grad` trong gradient descent.

```

def softmarginSVM_gd(X, y, w0, b0, eta):
    w = w0
    b = b0
    it = 0
    while it < 10000:
        it = it + 1
        (gw, gb) = grad(X, y, w, b)
        w -= eta*gw
        b -= eta*gb
        if (it % 1000) == 0:
            print('iter %d' %it + ' loss: %f' %loss(X, y, w, b))
    return (w, b)

w0 = .1*np.random.randn(X.shape[1])
b0 = .1*np.random.randn()
lr = 0.05
(w_hinge, b_hinge) = softmarginSVM_gd(X, y, w0, b0, lr)
print('w_hinge = ', w_dual)
print('b_hinge = ', b_dual)

```

Hình 27.4: Các đường phân chia tìm được bởi ba cách khác nhau: a) Thư viện sklearn, b) Giải bài toán đối ngẫu bằng CVXOPT, c) Hàm hinge loss. Các kết quả tìm được gần như giống nhau.

Kết quả:

```

iter 1000 loss: 0.436460
iter 2000 loss: 0.405307
iter 3000 loss: 0.399860
iter 4000 loss: 0.395440
iter 5000 loss: 0.394562
iter 6000 loss: 0.393958
iter 7000 loss: 0.393805
iter 8000 loss: 0.393942
iter 9000 loss: 0.394005
iter 10000 loss: 0.393758
w_hinge = [-1.87457279 -1.80695039]
b_hinge = 8.49672109814

```

Ta thấy rằng **loss** giảm dần và hội tụ theo thời gian, chứng tỏ *learning rate* là phù hợp. Nghiệm tìm được cũng gần giống nghiệm của hai cách làm phía trên.

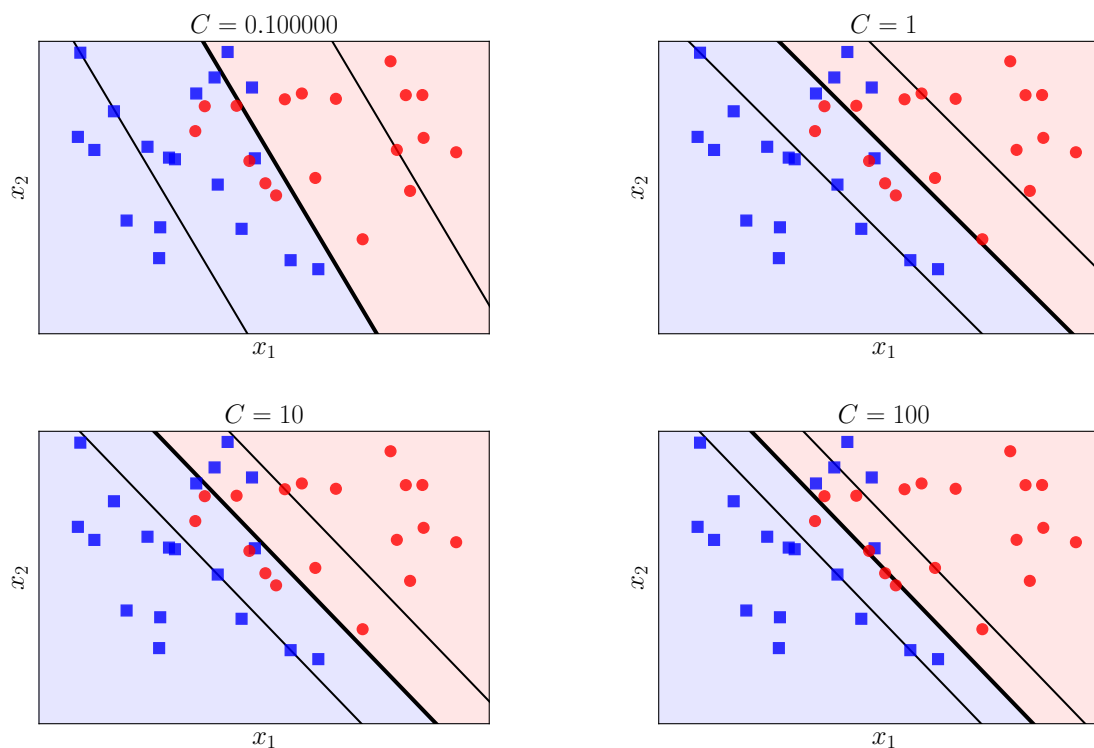
Hình 27.4 minh họa các nghiệm tìm được bằng ba phương pháp phía trên. Ta thấy rằng các nghiệm tìm được gần như giống nhau.

27.5.4 Ảnh hưởng của C lên nghiệm

Hình 27.5 minh họa nghiệm tìm được bằng sklearn với các giá trị C khác nhau. Quan sát thấy khi C càng lớn, biên càng nhỏ đi. Điều này phù hợp với các suy luận ở đầu chương.

27.6 Tóm tắt và thảo luận

- SVM thuần (*hard-margin SVM*) hoạt động không hiệu quả khi có nhiễu ở gần ranh giới 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 *margin* tương ứng. Bài toán tối



Hình 27.5: Ảnh hưởng của C lên nghiệm của soft-margin SVM. C càng lớn, biên càng nhỏ, và ngược lại.

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.

- 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 một cách hiệu quả bằng các phương pháp gradient descent.
- Soft-margin SVM yêu cầu chọn hằng số 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 [SSWB00].
- Source code cho chương này có thể được tìm thấy tại <https://goo.gl/PuWxba>.
- LIBSVM là một thư viện SVM phổ biến (<https://goo.gl/Dt7o7r>).
- **Đọc thêm:** L. Rosasco *et al.*, *Are Loss Functions All the Same?* (<https://goo.gl/QH2Cgr>). *Neural Computation*.2004 [RDVC+04].

Kernel support vector machine

28.1 Giới thiệu

Có một sự tương ứng thú vị giữa hai nhóm thuật toán phân lớp phổ biến nhất, neural network và support vector machine. Chúng đều bắt đầu từ bài toán phân lớp với hai lớp dữ liệu *linearly separable*, tiếp theo đến hai lớp *gần như linear separable*, đến bài toán với nhiều lớp dữ liệu, rồi các bài toán với các lớp hoàn toàn không *linearly separable*. Sự tương ứng được cho trong Bảng 28.1

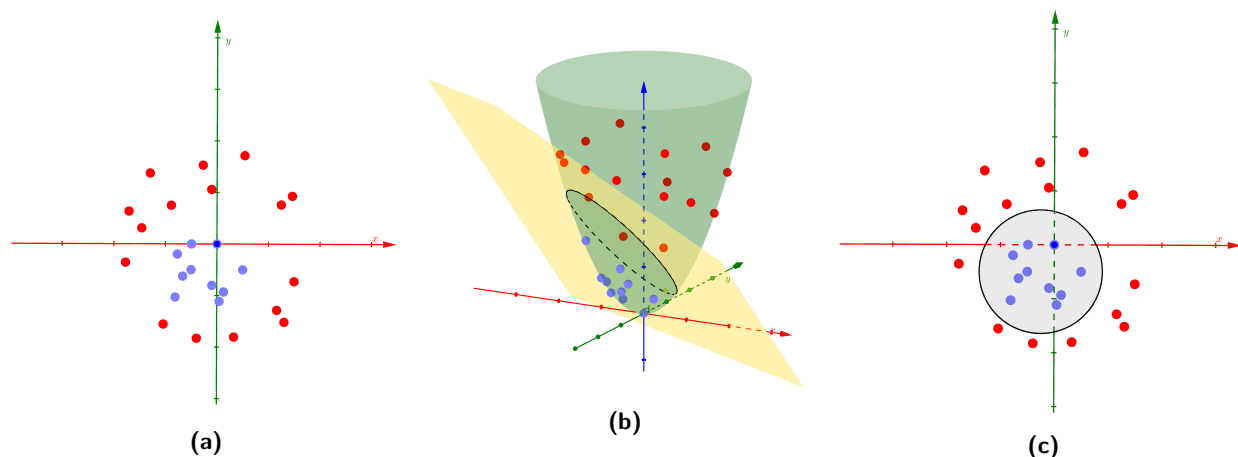
Bảng 28.1: Sự tương đồng giữa neural network và support vector machine

Neural network	Support vector machine	Tính chất chung
PLA	Hard-margin SVM	Hai lớp là <i>linearly separable</i>
Logistic regression	Soft-margin SVM	Hai lớp là <i>gần linearly separable</i>
Softmax regression	Multi-class SVM	Nhiều lớp dữ liệu (ranh giới là các siêu phẳng)
Multi-layer perceptron	Kernel SVM	Bài toán phân lớp với biên không <i>linearly separable</i>

Trong chương này, chúng ta cùng thảo luận về Kernel SVM, tức việc áp dụng SVM lên bài toán mà dữ liệu giữa hai lớp là hoàn toàn *không linear separable*. Bài toán phân biệt nhiều lớp dữ liệu sẽ được thảo luận trong chương tiếp theo.

Ý tưởng cơ bản của Kernel SVM và các phương pháp kernel nói chung là tìm một phép biến đổi dữ liệu *không linearly separable* ở một không gian sang một không gian mới. Ở không gian mới này, dữ liệu trở nên *linearly separable* hoặc *gần linearly separable*, và vì vậy, bài toán phân lớp có thể được giải quyết bằng hard/soft-margin SVM.

Xét ví dụ trên Hình 28.1 với việc biến dữ liệu không *linearly separable* trong không gian hai chiều thành *linearly separable* trong không gian ba chiều bằng cách giới thiệu thêm một



Hình 28.1: Ví dụ về Kernel SVM. (a) Dữ liệu của hai lớp 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 một mặt parabolic và đã hai lớp trở nên *linearly separable*. Mặt phẳng màu vàng là mặt phân chia, có thể tìm được bởi một 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 lớp.

chiều mới. Để xem ví dụ này một cách sinh động hơn, bạn có thể xem clip đi kèm với blog *Machine Learning cơ bản* tại <https://goo.gl/3wMHyZ>.

Nói một cách toán học, kernel SVM là phương pháp đi tìm một hàm số biến đổi dữ liệu \mathbf{x} từ không gian đặc trưng ban đầu thành dữ liệu trong một không gian mới bằng một hàm số $\Phi(\mathbf{x})$. Trong ví dụ này, hàm $\Phi(\cdot)$ đơn giản là giới thiệu thêm một chiều dữ liệu mới là một hàm số của các thành phần đặc trưng đã biết. Hàm số này cần thỏa mãn: trong không gian mới, dữ liệu giữa hai lớp là *linearly separable* hoặc *gần như linearly separable*. Khi đó, ta có thể dùng các bộ phân lớp tuyến tính thông thường như PLA, logistic regression, hay hard/soft margin SVM.

Các hàm $\Phi(\mathbf{x})$ 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 hàm *kernel* mô tả quan hệ giữa hai điểm dữ liệu bất kỳ trong không gian mới, thay vì đi tính toán trực tiếp biến đổi của từng điểm dữ liệu trong không gian mới. Kỹ thuật này được xây dựng dựa trên quan sát về các bài toán đối ngẫu của hard/soft margin SVM.

Nếu phải so sánh, ta có thể thấy rằng các hàm *kernel* có chức năng tương tự như các hàm *activation* trong neural network vì chúng đều giúp giải quyết các bài toán với dữ liệu *linearly separable*. Trong Mục 28.2, chúng ta cùng tìm hiểu cơ sở toán học của Kernel SVM, Mục 28.3 sẽ giới thiệu một số hàm *kernel* thường được sử dụng.

28.2 Cơ sở toán học

Cùng nhắc lại bài toán tối ưu ngẫu nhiên trong soft-margin SVM cho dữ liệu *gần linearly separable*:

$$\begin{aligned} \boldsymbol{\lambda} &= \arg \max_{\boldsymbol{\lambda}} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \\ \text{thỏa mãn: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \end{aligned} \quad (28.1)$$

trong đó, N là số cặp điểm dữ liệu trong tập huấn luyện; \mathbf{x}_n là vector đặc trưng của dữ liệu thứ n trong tập training; y_n là *nhãn* của điểm dữ liệu thứ n , bằng 1 hoặc -1; λ_n là nhân tử Lagrange ứng với điểm dữ liệu thứ n ; và C là một 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 (28.1), *nhãn* của một điểm dữ liệu mới sẽ được xác định bởi

$$\text{class}(\mathbf{x}) = \text{sgn} \left\{ \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \right\} \quad (28.2)$$

trong đó, $\mathcal{M} = \{n : 0 < \lambda_n < C\}$ là tập hợp những điểm nằm trên các *margin*; $\mathcal{S} = \{n : 0 < \lambda_n\}$ là tập hợp các *support vector*; và $N_{\mathcal{M}}$ 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 (28.1) có thể không thực sự tạo ra một bộ phân lớp tốt. Giả sử rằng ta có thể tìm được hàm số $\Phi(\cdot)$ 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 (28.1) trở thành:

$$\begin{aligned} \boldsymbol{\lambda} &= \arg \max_{\boldsymbol{\lambda}} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x}_m) \\ \text{thỏa mãn: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \end{aligned} \quad (28.3)$$

và *nhãn* của một điểm dữ liệu mới được xác định bởi dấu của biểu thức

$$\mathbf{w}^T \Phi(\mathbf{x}) + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}_n) \right) \quad (28.4)$$

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 *nhãn* 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 (28.3) và biểu thức (28.4), 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. Vì vậy, ta có thể không cần xác định hàm $\Phi(\cdot)$ mà chỉ cần xác định một hàm $k(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$. 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 đi 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 (28.3) và biểu thức (28.4) như sau:

$$\begin{aligned} \boldsymbol{\lambda} &= \arg \max_{\boldsymbol{\lambda}} \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m k(\mathbf{x}_n, \mathbf{x}_m) \\ \text{thoả mãn: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \end{aligned} \tag{28.5}$$

và

$$\sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}_n) \right) \tag{28.6}$$

Ví dụ: Xét phép biến đổi một 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:

$$\Phi(\mathbf{x})^T \Phi(\mathbf{z}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2][1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, \sqrt{2}z_1z_2, z_2^2]^T \tag{28.7}$$

$$= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2 \tag{28.8}$$

$$= (1 + x_1z_1 + x_2z_2)^2 = (1 + \mathbf{x}^T \mathbf{z})^2 = k(\mathbf{x}, \mathbf{z}) \tag{28.9}$$

Trong ví dụ này, rõ ràng rằng việc tính toán hàm kernel $k(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^T \mathbf{z})^2$ cho hai điểm dữ liệu dễ dàng hơn việc tính từng $\Phi(\cdot)$ rồi nhân chúng với nhau. Hơn nữa, giá trị thu được là một số vô hướng thay vì phải lưu hai vector năm chiều $\Phi(\mathbf{x}), \Phi(\mathbf{z})$.

Vậy những hàm số kernel cần có những tính chất gì, và những hàm như thế nào được sử dụng trong thực tế?

28.3 Hàm số kernel

28.3.1 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:

- Đối xứng: $k(\mathbf{x}, \mathbf{z}) = k(\mathbf{z}, \mathbf{x})$, vì tích vô hướng của hai vector có tính đối xứng.
- Về lý thuyết, hàm kernel cần thỏa mãn điều kiện Mercer¹:

$$\sum_{n=1}^N \sum_{m=1}^N k(\mathbf{x}_m, \mathbf{x}_n) c_n c_m \geq 0, \quad \forall c_i \in \mathbb{R}, i = 1, 2, \dots, N \quad (28.10)$$

Với mọi tập hữu hạn các vector $\mathbf{x}_1, \dots, \mathbf{x}_n$. Tính chất này để đảm bảo cho việc hàm mục tiêu của bài toán đối ngẫu (28.5) là lồi. Thật vậy, nếu một hàm kernel thỏa mãn điều kiện (28.10), xét $c_n = y_n \lambda_n$, ta sẽ có:

$$\boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} = \sum_{n=1}^N \sum_{m=1}^N k(\mathbf{x}_m, \mathbf{x}_n) y_n y_m \lambda_n \lambda_m \geq 0, \quad \forall \lambda_n \quad (28.11)$$

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 (28.11) ta suy ra \mathbf{K} là một ma trận nửa xác định dương. Vì vậy, bài toán tối ưu (28.5) 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 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, chúng ta chỉ tập trung vào các hàm kernel thông dụng và có sẵn trong các thư viện.

Việc giải quyết bài toán (28.5) hoàn toàn tương tự như bài toán đối ngẫu của soft-margin SVM, chúng ta sẽ không bàn tới trong chương này. Thay vào đó, các hàm kernel thông dụng và hiệu năng của chúng trong các bài toán thực tế sẽ được thảo luận. Việc này sẽ được thực hiện thông qua các ví dụ và cách sử dụng thư viện sklearn.

28.3.2 Một số hàm kernel thông dụng

Linear

Đây 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 Chương 26, thỏa mãn điều kiện (28.10).

Khi sử dụng `sklearn.svm.SVC`, kernel này được chọn bằng cách chọn `kernel = 'linear'`.

¹ Xem *Kernel method*–Wikipedia (<https://goo.gl/YXct7F>)

Polynomial

Hàm kernel của polynomial có dạng

$$k(\mathbf{x}, \mathbf{z}) = (r + \gamma \mathbf{x}^T \mathbf{z})^d \quad (28.12)$$

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 được dùng để mô tả hầu hết các đa thức có bậc không vượt quá d nếu d là một số tự nhiên.

Khi sử dụng thư viện `sklearn`, kernel này được chọn bằng cách đặt `kernel = 'poly'`. Bạn đọc được khuyến khích đọc tài liệu chính thức trong scikit-learn tại <https://goo.gl/QtFc9>.

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

$$k(\mathbf{x}, \mathbf{z}) = \exp(-\gamma \|\mathbf{x} - \mathbf{z}\|_2^2), \quad \gamma > 0 \quad (28.13)$$

Trong `sklearn`, kernel này được lựa chọn bằng cách đặt `kernel = 'rbf'`.

Sigmoid

Hàm dạng sigmoid cũng được sử dụng làm kernel, với

$$k(\mathbf{x}, \mathbf{z}) = \tanh(\gamma \mathbf{x}^T \mathbf{z} + r) \quad (28.14)$$

Trong `sklearn`, kernel này được lựa chọn bằng `kernel = 'sigmoid'`.

Bảng tóm tắt các kernel thông dụng

Bảng 28.2 tóm tắt các kernel thông dụng và cách sử dụng chúng trong `sklearn`.

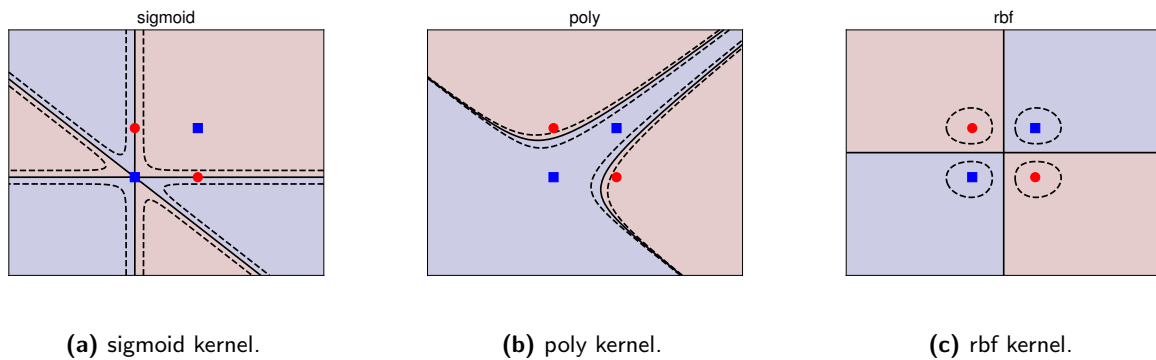
Bảng 28.2: Bảng các kernel thông dụng

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 : degree , γ : gamma , r : coef0
'sigmoid'	$\tanh(\gamma \mathbf{x}^T \mathbf{z} + r)$	γ : gamma , r : coef0
'rbf'	$\exp(-\gamma \ \mathbf{x} - \mathbf{z}\ _2^2)$	$\gamma > 0$: gamma

Nếu bạn muốn sử dụng các thư viện cho C/C++, các bạn có thể tham khảo LIBSVM (<https://goo.gl/Dt7o7r>) và LIBLINEAR (<https://goo.gl/ctD7a3>).

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 tại <https://goo.gl/A9ajzp>.



Hình 28.2: 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 (28.6) bằng 0. Các đường nét đứt là các đường đồng mức ứng với giá trị của biểu thức (28.6) 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.

28.4 Ví dụ minh họa

28.4.1 Bài toán XOR

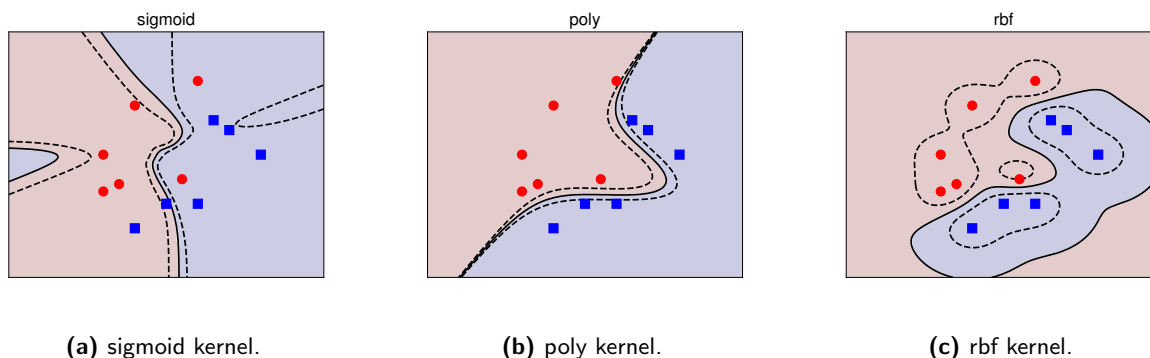
Chúng ta cùng quay lại với bài toán XOR. Chúng ta biết rằng bài toán XOR không thể giải quyết nếu chỉ dùng một bộ phân lớp tuyến tính. Chúng ta cùng giải quyết bài toán này bằng SVM với các kernel khác nhau. Kết quả được minh họa trong Hình 28.2.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# XOR dataset and targets
X = np.array([[0, 0], [1, 1], [1, 0], [0, 1]])
y = np.array([0, 0, 1, 1])
# fit the model
for kernel in ('sigmoid', 'poly', 'rbf'):
    clf = svm.SVC(kernel=kernel, gamma=4, coef0 = 0)
    clf.fit(X, y)
```

Ta có các nhận xét đối với mỗi kernel như sau:

- **sigmoid:** nghiệm tìm được không thật tốt vì có ba trong bốn điểm nằm chính xác trên đường phân chia. Nói cách khác, nghiệm này sẽ rất *nhạy cảm với nhiễu*.
- **poly:** Nghiệm này có tốt hơn nghiệm của **sigmoid** nhưng kết quả có phần *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 lớp. 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`.



Hình 28.3: 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.

28.4.2 Dữ liệu gần linearly separable

Xét một ví dụ khác với dữ liệu giữa hai lớp là *gần linearly separable* như Hình 28.3.

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.

28.4.3 Kernel SVM cho MNIST

Tiếp theo, chúng ta cùng làm một thí nghiệm nhỏ bằng cách áp dụng SVM với RBF kernel vào bài toán phân loại 4 chữ số `0`, `1`, `2`, `3` của tập MNIST. Trước hết, chúng ta cần lấy ra dữ liệu thuộc các chữ số này. Dữ liệu được chuẩn hoá về đoạn $[0, 1]$ bằng cách chia toàn bộ các thành phần cho 255 (giá trị cao nhất của mỗi pixel)

```
from __future__ import print_function
import numpy as np
from sklearn import svm
from sklearn.datasets import fetch_mldata
data_dir = '../..data' # path to your data folder
mnist = fetch_mldata('MNIST original', data_home=data_dir)

X_all = mnist.data/255. # data normalization
y_all = mnist.target
digits = [0, 1, 2, 3]
ids = []
for d in digits:
    ids.append(np.where(y_all == d)[0])

selected_ids = np.concatenate(ids, axis = 0)
X = X_all[selected_ids]
y = y_all[selected_ids]
print('Number of samples = ', X.shape[0])
```

Kết quả:

```
Number of samples = 28911
```

Như vậy, có khoảng 29000 điểm dữ liệu tổng cộng. Chúng ta lấy ra 24000 điểm làm tập kiểm thử, còn lại là tập huấn luyện. Bộ phân lớp kernel SVM với RBF sẽ được sử dụng.

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size= 24000)

model = svm.SVC(kernel='rbf', gamma=.1, coef0 = 0)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print("Accuracy: %.2f %%" % (100*accuracy_score(y_test, y_pred)))
```

Kết quả:

```
Accuracy: 94.22 %
```

Kết quả thu được là khoảng 94%. Nếu chọn nhiều điểm dữ liệu để huấn luyện và thay đổi các tham số **gamma**, **coef0**, bạn đọc có thể sẽ thu được các kết quả tốt hơn. Đây là một bài toán *multi-class* classification, và cách giải quyết của thư viện này là *one-vs-rest*. Như đã đề cập trong Chương 14, *one-vs-rest* có nhiều hạn chế vì phải huấn luyện nhiều bộ phân lớp. Hơn nữa, với kernel SVM, việc tính toán các kernel cũng trở nên phức tạp khi lượng dữ liệu và số chiều dữ liệu tăng lên.

28.5 Tóm tắt

- Trong bài toán phân lớp nhị phân, nếu dữ liệu của hai lớp là *không linearly section*, 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 (*gần*) *linearly separable*.
- Việc tính toán trực tiếp hàm $\Phi()$ đôi khi 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 bốn 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.
- Source code cho chương này có thể được tìm thấy tại <https://goo.gl/6sbds5>.

Multi-class support vector machine

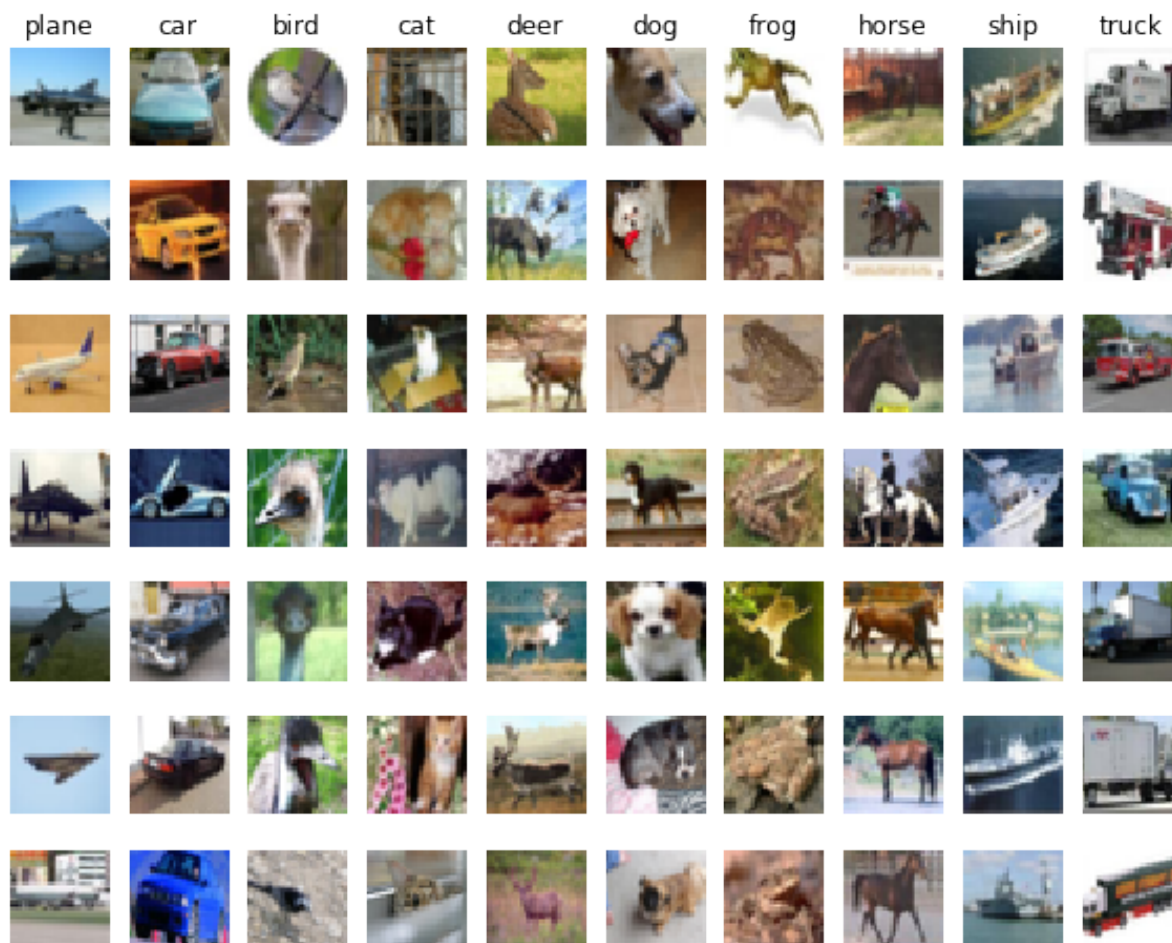
29.1 Giới thiệu

29.1.1 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 lớp dữ liệu ($C = 2$). Một cách tự nhiên để mở rộng các mô hình này áp dụng cho các bài toán *multi-class classification*, tức có nhiều lớp dữ liệu khác nhau ($C > 2$), là sử dụng nhiều *binary classifier* và các kỹ thuật như *one-vs-one* hoặc *one-vs-rest*. Cách làm này có những hạn chế như đã trình bày trong Chương 14.

Softmax regression (xem Chương 15), là một phương pháp tổng quát của *logistic regression*, được sử dụng phổ biến nhất trong các mô hình phân lớp hiện nay. Về cơ bản, thuật toán huấn luyện softmax regression đi tìm ma trận hệ số $\mathbf{W} \in \mathbb{R}^{d \times C}$ và vector bias $\mathbf{b} \in \mathbb{R}^C$ sao cho với một điểm dữ liệu được mô tả bởi một vector đặc trưng d chiều, vector $\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$ có thành phần lớn nhất nằm ở chỉ số tương ứng với nhãn chính xác của \mathbf{x} . Vector \mathbf{z} , còn được gọi là *score vector*, tiếp tục được đưa qua hàm *softmax* để ước lượng xác suất để điểm dữ liệu \mathbf{x} rơi vào mỗi lớp.

Trong chương này, chúng ta sẽ thảo luận một phương pháp phổ biến khác cũng được dùng cho các bài toán *multi-class classification* có tên là *multi-class SVM*. Trong đó, ta cũng phải đi tìm ma trận hệ số \mathbf{W} và vector bias \mathbf{b} sao cho với một điểm dữ liệu \mathbf{x} , vector $\mathbf{W}^T \mathbf{x} + \mathbf{b}$ cũng có thành phần cao nhất tại chỉ số tương ứng với nhãn của \mathbf{x} . Tuy nhiên, hàm mất mát để ép việc này xảy ra trên tập huấn luyện được xây dựng dựa trên hàm *hinge loss* (của softmax regression là *cross entropy loss*). Thuật toán tối ưu hàm mất mát này của *multi-class SVM* cũng dựa trên gradient descent. Và *multi-class SVM* cũng có thể được tích hợp vào layer cuối cùng của các neural network để tạo ra một bộ phân lớp khá hiệu quả.



Hình 29.1: Ví dụ về các bức ảnh trong 10 lớp của bộ dữ liệu CIFAR10.

Trong chương này, chúng ta sẽ tìm hiểu *multi-class SVM* thông qua một ví dụ về bài toán phân loại các bức ảnh thuộc 10 lớp khác nhau trong bộ cơ sở dữ liệu CIFAR10 (<https://goo.gl/9KKbQu>).

29.1.2 Bộ cơ sở dữ liệu CIFAR10

Bộ cơ sở dữ liệu CIFAR10 gồm 60000 ảnh khác nhau thuộc 10 lớp dữ liệu: *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 lớp được cho trong Hình 29.1. Tập huấn luyện bao gồm 50000 bức ảnh, tập kiểm thử bao gồm 10000 ảnh còn lại. Trong số 50000 ảnh huấn luyện, 1000 ảnh sẽ được lấy ra ngẫu nhiên để làm tập validation. Đây là một bộ cơ sở dữ liệu tương đối khó vì kích thước của các bức ảnh là nhỏ và các bức ảnh trong cùng một lớp 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 96% (<https://goo.gl/w1sgK4>), sử dụng một *convolutional neural network* nhiều layer kết hợp với softmax regression ở layer cuối cùng. Trong chương này, chúng ta sẽ sử dụng một mô hình neural network đơn giản không có hidden layer nào và layer cuối cùng là một

multi-class SVM để giải quyết bài toán. Độ chính xác đạt được là khoảng 40%, nhưng cũng là đã rất ấn tượng. Chúng ta cùng phân tích multi-class SVM và lập trình mà không sử dụng một thư viện đặc biệt nào ngoài numpy. Bài toán này cũng như nội dung chính của chương được lấy từ Lecture notes *Linear Classifier II – CS231n 2016* (<https://goo.gl/y3QsDP>) và *Assignment #1 – CS231n 2016* (<https://goo.gl/1Qh84b>).

Trước khi đi vào mục xây dựng và tối ưu hàm mất mát cho multi-class SVM, chúng ta cần làm một chút *feature engineering* để tạo ra vector đặc trưng cho mỗi ảnh. Cách làm này có thể được sử dụng kèm với các bộ phân lớp khác, không nhất thiết là chỉ multi-class SVM.

29.1.3 Xây dựng vector đặc trưng

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 *chuẩn hoá dữ liệu* (*data normalization*).

- Mỗi ảnh màu 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 channel Red, Green, Blue của bức ảnh ra thành một vector có kích thước là $3 \times 32 \times 32 = 3072$.
- 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 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 hoá dữ liệu trong *validation set* và *test set*.

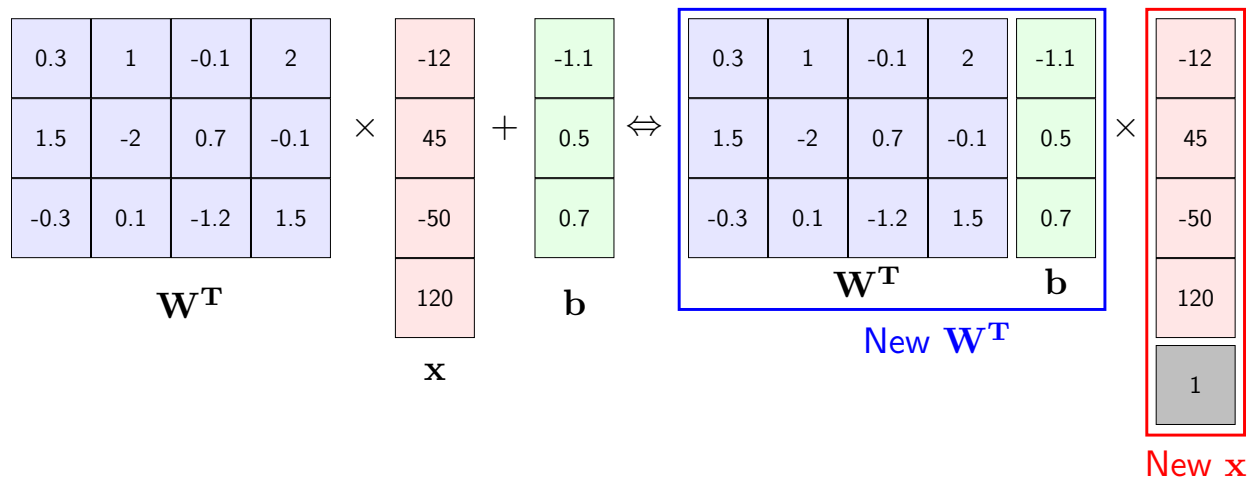
29.1.4 Bias trick

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:

$$f(\mathbf{x}, \mathbf{W}, \mathbf{b}) = \mathbf{W}^T \mathbf{x} + \mathbf{b} \quad (29.1)$$

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 29.2. 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 29.2.

Tiếp theo, chúng ta viết chương trình lấy dữ liệu từ tập CIFAR10, chuẩn hoá dữ liệu và thêm đặc trưng bằng 1 vào cuối mỗi vector. Đồng thời, 1000 dữ liệu từ tập huấn luyện cũng được tách ra làm tập validation.



Hình 29.2: Bias trick.

```

from __future__ import print_function
import numpy as np
# need cs231n folder from https://goo.gl/cgJgcG
from cs231n.data_utils import load_CIFAR10

# Load CIFAR 10 dataset
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# Extract a validation from X_train
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size= 1000)

# mean image of all training images
img_mean = np.mean(X_train, axis = 0)

def feature_engineering(X):
    X -= img_mean # zero-centered
    N = X.shape[0] # number of data point
    X = X.reshape(N, -1) # vectorization
    return np.concatenate((X, np.ones((N, 1))), axis = 1) # bias trick

X_train = feature_engineering(X_train)
X_val = feature_engineering(X_val)
X_test = feature_engineering(X_test)
print('X_train shape = ', X_train.shape)
print('X_val shape = ', X_val.shape)
print('X_test shape = ', X_test.shape)

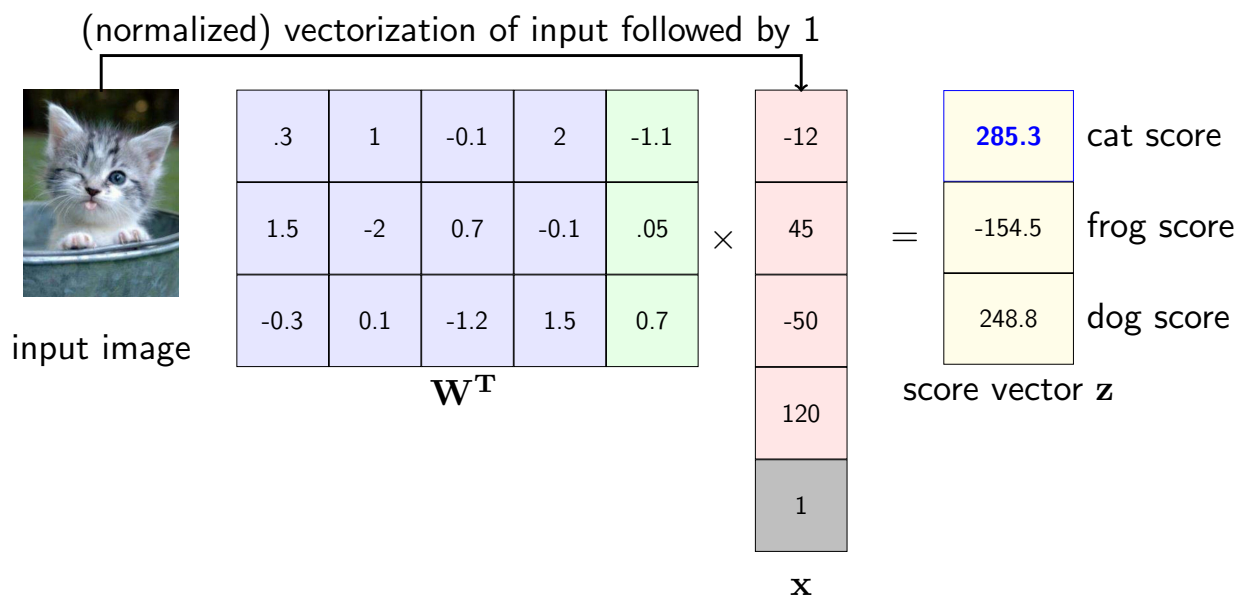
```

Kết quả:

```

X_train shape = (49000, 3073)
X_val shape = (1000, 3073)
X_test shape = (10000, 3073)

```

Hình 29.3: Ví dụ về cách tính score vector. Khi test, nhãn của dữ liệu được xác định dựa trên class có score cao nhất.

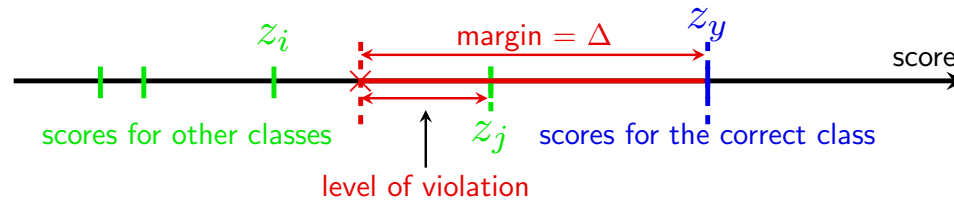
29.2 Xây dựng hàm mất mát

29.2.1 Hinge losses tổng quát cho multi-class SVM

Trong multi-class SVM, khi kiểm thử, nhãn của một điểm dữ liệu mới được xác định bởi thành phần có giá trị lớn nhất trong score vector $\mathbf{z} = \mathbf{W}^T \mathbf{x}$ (xem Hình 29.3). Điều này giống với softmax regression. Softmax regression sử dụng *cross-entropy loss* để ép hai vector xác suất bằng nhau, tức ép phần tử tương ứng với *nhãn đúng* (*correct class*) trong vector xác suất gần với 1, đồng thời khiến 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 *biên an toàn*, giống như trong hard/soft-margin SVM với hai lớp dữ liệu. Multi-class SVM ép 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 *biên an toàn*, như được mô tả trong Hình 29.4.

Nếu score tương ứng với *correct class* lớn hơn các score khác một khoảng bằng một *biên an toàn* Δ thì *không có mất mát nào xảy ra*, tức sự mất mát bằng 0. Nói cách khác, những score nằm ở bên trái điểm \times màu đỏ không gây ra mất mát nào. Ngược lại, các điểm có score nằm phía phải của điểm \times sẽ bị *xử phạt*, và càng vi phạm nhiều sẽ bị xử lý ở mức càng 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 lớp dữ liệu 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



Hình 29.4: Mô tả hinge loss trong multi-class SVM. Multi-class SVM ép score của *correct class*, được minh họa bởi điểm màu lam, cao hơn các score khác, minh họa bởi các điểm màu lục, một khoảng cách an toàn Δ là đoạn màu đỏ. Những score khác nằm trong vùng an toàn (phía trái của điểm \times 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.

$\mathbf{z} = \mathbf{W}^T \mathbf{x}$. Như vậy, score của *correct class* là z_y , score của các lớp khác là các $z_i, i \neq y$. Trong Hình 29.4, các score z_i nằm trong vùng an toàn và z_j trong vùng vi phạm. Với mỗi score z_i trong vùng an toàn, *loss* bằng 0. Với mỗi score z_j vượt quá điểm an toàn (điểm \times), *loss* do nó gây ra được tính bằng lượng vượt quá so với điểm \times đó, 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

$$\max(0, \Delta - z_y + z_j) = \max(0, \Delta - \mathbf{w}_y^T \mathbf{x} + \mathbf{w}_j^T \mathbf{x}) \quad (29.2)$$

trong đó \mathbf{w}_j là cột thứ j của ma trận hệ số \mathbf{W} . Như vậy, với một điểm dữ liệu $\mathbf{x}_n, n = 1, 2, \dots, N$ với nhãn y_n , tổng cộng *loss* do nó gây ra là

$$\mathcal{L}_n = \sum_{j \neq y_n} \max(0, \Delta - z_{y_n}^n + z_j^n)$$

với $\mathbf{z}^n = \mathbf{W}^T \mathbf{x}_n = [z_1^n, z_2^n, \dots, z_C^n]^T \in \mathbb{R}^{C \times 1}$ là score vector tương ứng với điểm dữ liệu \mathbf{x}_n . Với toàn bộ các điểm dữ liệu $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]$, *loss* được định nghĩa là

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, \Delta - z_{y_n}^n + z_j^n) \quad (29.3)$$

với $\mathbf{y} = [y_1, y_2, \dots, y_N]$ là vector chứa *correct class* của toàn bộ các điểm trong training set.

29.2.2 Regularization

Điều gì sẽ xảy ra nếu nghiệm tìm được \mathbf{W} là một nghiệm *hoàn hảo*, tức không có score nào *vi phạm* và hàm mất mát (29.3) 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 tiến tới vô cùng khiến cho bài toán

rất *không ổn định* (*unstable*) khi tối ưu. 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

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \underbrace{\frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, \Delta - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n)}_{\text{data loss}} + \underbrace{\frac{\lambda}{2} \|\mathbf{W}\|_F^2}_{\text{regularization loss}} \quad (29.4)$$

với λ là một giá trị dương giúp cân bằng giữa *data loss* và *regularization loss*, thường được chọn bằng cross-validation.

29.2.3 Hàm mất mát của multi-class SVM

Có hai *hyperparameter* trong hàm mất mát (29.4) 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? Trên thực tế, người ta nhận thấy rằng Δ có thể được chọn bằng 1 mà không ảnh hưởng nhiều tới chất lượng của nghiệm (<https://goo.gl/NSyfQi>). Từ đó, hàm mất mát cuối cùng cho multi-class SVM có dạng

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) + \frac{\lambda}{2} \|\mathbf{W}\|_F^2 \quad (29.5)$$

Một lần nữa, chúng ta có thể dùng gradient descent để tìm nghiệm cho bài toán tối ưu không ràng buộc này. Việc này sẽ được thảo luận kỹ trong Mục 29.3.

29.2.4 Soft-margin SVM là một trường hợp đặc biệt của multi-class SVM

Điều này có thể được nhận ra bằng cách xét từng điểm dữ liệu. Trong (29.5), nếu số lớp dữ liệu $C = 2$, tạm bỏ qua *regularization loss*, hàm mất mát tại mỗi điểm dữ liệu trở thành

$$\mathcal{L}_n = \sum_{j \neq y_n} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) \quad (29.6)$$

Xét hai trường hợp:

- $y_n = 1 \Rightarrow \mathcal{L}_n = \max(0, 1 - \mathbf{w}_1^T \mathbf{x}_n + \mathbf{w}_2^T \mathbf{x}_n) = \max(0, 1 - (\mathbf{w}_1 - \mathbf{w}_2)^T \mathbf{x})$
- $y_n = 2 \Rightarrow \mathcal{L}_n = \max(0, 1 - \mathbf{w}_2^T \mathbf{x}_n + \mathbf{w}_1^T \mathbf{x}_n) = \max(0, 1 - (-1)(\mathbf{w}_1 - \mathbf{w}_2)^T \mathbf{x})$

Nếu ta thay $y_n = -1$ cho dữ liệu thuộc lớp có nhãn bằng 2, và đặt $\bar{\mathbf{w}} = \mathbf{w}_1 - \mathbf{w}_2$, hai trường hợp trên có thể được viết gọn thành

$$\mathcal{L}_n = \max(0, 1 - y_n \bar{\mathbf{w}}^T \mathbf{x}_n)$$

Đây chính là hinge loss cho soft-margin SVM.

29.3 Tính toán hàm mất mát và đạo hàm của nó

Vì hàm mất mát của multi-class SVM *hơi phức tạp một chút*, đạo hàm của nó theo \mathbf{W} cũng khó có thể được suy ra dễ dàng. Chúng ta cần kiểm tra liệu đạo hàm tính được có thực sự chính xác không trước khi thực hiện gradient descent. Phương pháp quen thuộc được sử dụng là tính *numerical gradient descent*. Để thực hiện phương pháp này, chúng ta cũng cần tính giá trị của hàm mất mát tại một điểm \mathbf{W} bất kỳ.

Việc tính toán giá trị của hàm mất mát và đạo hàm của nó tại \mathbf{W} bất kỳ không những cần sự chính xác mà còn cần được thực hiện một cách hiệu quả. Để đạt được việc này, chúng ta sẽ làm từng bước một. Bước thứ nhất là đảm bảo rằng các tính toán là *chính xác*, dù cách tính có thể rất chậm. Bước thứ hai, ta phải đảm bảo có một cách tính *hiệu quả* để thuật toán chạy nhanh hơn. Hai bước này nên được thực hiện trên một lượng dữ liệu nhỏ để có thể nhanh chóng thấy được kết quả. Việc tính *numerical gradient* trên dữ liệu lớn thường tốn rất nhiều thời gian. Các quy tắc này cũng được áp dụng với các bài toán tối ưu khác có sử dụng đạo hàm trong quá trình tìm nghiệm.

Hai mục tiếp theo sẽ mô tả hai bước đã nêu ở trên.

29.3.1 Tính hàm mất mát và đạo hàm một cách chính xác

Dưới đây là cách tính đơn giản cho hàm mất mát và đạo hàm trong (29.5) với hai vòng `for`. Chú ý thành phần *regularization*.

```
def svm_loss_naive(W, X, y, reg):
    ''' calculate loss and gradient of the loss function at W. Naive way
    W: 2d numpy array of shape (d, C). The weight matrix.
    X: 2d numpy array of shape (N, d). The training data
    y: 1d numpy array of shape (N,). The training label
    reg: a positive number. The regularization parameter
    '''
    d, C, N = W.shape, X.shape[0] # data dim, number of classes, number of points
    loss = 0
    dW = np.zeros_like(W)
    for n in xrange(N):
        xn = X[n]
        score = xn.dot(W)
        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)
    dW /= N
    dW += reg*W
    return loss, dW ## continue on next page
```

```

# random, small data
d, C, N = 100, 3, 300
reg = .1
W_rand = np.random.randn(d, C)
X_rand = np.random.randn(N, d)
y_rand = np.random.randint(0, C, N)

# sanity check
print('Loss with reg = 0 :', svm_loss_naive(W_rand, X_rand, y_rand, 0)[0])
print('Loss with reg = 0.1:', svm_loss_naive(W_rand, X_rand, y_rand, .1)[0])

```

Kết quả:

```

Loss with reg = 0 : 12.5026818221
Loss with reg = 0.1: 27.7805360552

```

Cách tính với hai vòng **for** lồng nhau như trên mô tả lại chính xác biểu thức (29.5) nên sai sót, nếu có, 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.

Cách tính đạo hàm cho phần *data loss* phía trên dựa trên nhận xét sau đây:

$$\nabla_{\mathbf{w}_{y_n}} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) = \begin{cases} 0 & \text{nếu } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n < 0 \\ -\mathbf{x}_n & \text{nếu } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n > 0 \end{cases} \quad (29.7)$$

$$\nabla_{\mathbf{w}_j} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) = \begin{cases} 0 & \text{nếu } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n < 0 \\ \mathbf{x}_n & \text{nếu } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n > 0 \end{cases} \quad (29.8)$$

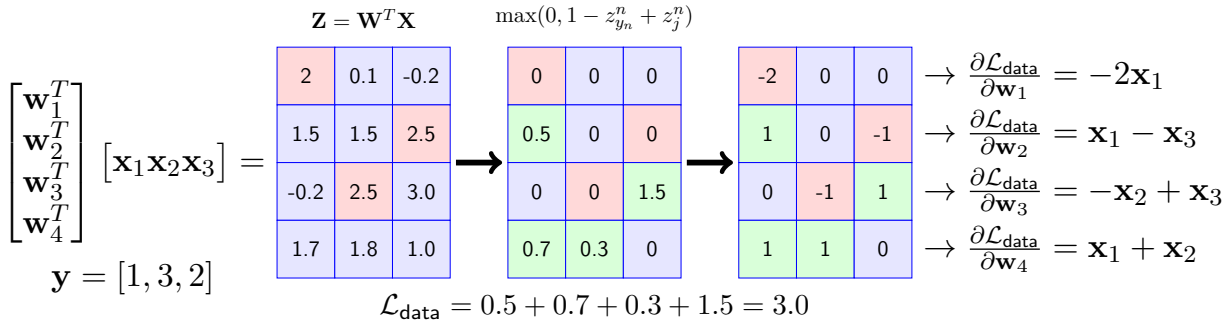
Rõ ràng là các đạo hàm này không xác định tại các điểm mà $1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n = 0$. Tuy nhiên, khi thực hành, ta có thể giả sử rằng tại 0, các đạo hàm này cũng bằng 0.

Để kiểm tra lại cách tính đạo hàm như trên dựa vào (29.7) và (29.8) có chính xác không, chúng ta cần 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. Bạn đọc có thể tự coi đây như một bài tập nhỏ.

Khi sự khác nhau giữa hai cách tính đạo hàm là nhỏ, 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.

29.3.2 Tính hàm mất mát và đạo hàm một cách hiệu quả

Các cách tính hiệu quả thường không chứa các vòng **for** mà được viết gọn lại dưới dạng ma trận và vector (*vectorization*). Để dễ hình dung, chúng ta cùng quan sát Hình 29.5. Ở đây, chúng ta tạm quên phần *regularization loss* đi vì cả *loss* và đạo hàm của phần này đều có cách tính đơn giản. Với phần *data loss*, chúng ta cũng bỏ qua hệ số $\frac{1}{N}$.



Hình 29.5: Mô phỏng cách tính giá trị và đạo hàm của hàm mất mát trong multi-class SVM.

Giả sử rằng có bốn lớp dữ liệu và mini-batch \mathbf{X} gồm có ba điểm dữ liệu $\mathbf{X} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \mathbf{x}_3]$. Ba điểm này lần lượt thuộc vào các lớp 1, 3, 2 (vector \mathbf{y}). 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* $Z = W^T X$.
- **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 và có thể coi chúng bằng 0 vì biểu thức *data loss* không chứa thành phần $j = y_n$. 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*. Ví dụ, nhìn vào ma trận màu ở giữa, giá trị ở hàng thứ hai, cột thứ nhất bằng $\max(0, 1 - 2 + 1.5) = \max(0, .5) = .5$. Giá trị ở hàng thứ ba, cột thứ nhất bằng $\max(0, 1 - 2 + (-0.2)) = \max(0, -1.2) = 0$. Giá trị ở hàng thứ tư, cột thứ nhất bằng $\max(0, 1 - 2 + 1.7) = 0.7$. Tương tự như thế với các cột còn lại.
- **Bước 3:** Theo công thức (29.7) và (29.8), với ô màu lục ở hàng thứ hai, cột thứ nhất (ứng với điểm dữ liệu \mathbf{x}_1), 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 bị trừ đi một lượng \mathbf{x}_1 . Như vậy, trong cột thứ nhất, có bao nhiêu ô màu lục thì có bấy nhiêu lần đạo hàm của \mathbf{w}_1 bị trừ đi một lượng \mathbf{x}_1 . Xét ma trận màu bên phải, giá trị ở ô trong hàng thứ i , cột thứ j là hệ số của đạo hàm theo \mathbf{w}_i gây ra bởi điểm dữ liệu \mathbf{x}_j . Tất cả các ô màu lục đều có giá trị bằng 1. Ô màu đỏ ở cột thứ nhất phải bằng -2 vì cột đó có hai ô màu lục. Tương tự với các ô màu lục và đỏ còn lại.
- **Bước 4:** Bây giờ cộng theo các hàng, ta sẽ được đạo hàm theo hệ số của lớp 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[0]
    loss = 0
    dW = np.zeros_like(W)

    Z = X.dot(W) # shape of (N, C)
    id0 = np.arange(Z.shape[0])
    correct_class_score = Z[id0, y].reshape(N, 1) # shape of (N, 1)
    margins = np.maximum(0, Z - correct_class_score + 1) # shape of (N, C)
    margins[id0, y] = 0
    loss = np.sum(margins)
    loss /= N
    loss += 0.5 * reg * np.sum(W * W)

    F = (margins > 0).astype(int) # shape of (N, C)
    F[np.arange(F.shape[0]), y] = np.sum(-F, axis = 1)
    dW = X.T.dot(F)/N + reg*W
    return loss, dW

```

Đoạn code phía trên không chứa vòng **for** nào. Để kiểm tra tính chính xác và hiệu quả của hàm này, chúng ta cần kiểm chứng ba điều. (i) Giá trị hàm mất mát đã chính xác chưa. (ii) Giá trị đạo hàm đã chính xác chưa. (iii) Cách tính đã thực sự hiệu quả chưa. Ba điều này có thể được kiểm chứng thông qua đoạn code dưới đây.

```

d, C = 3073, 10
W_rand = np.random.randn(d, C)
import time
t1 = time.time()
l1, dW1 = svm_loss_naive(W_rand, X_train, y_train, reg)
t2 = time.time()
l2, dW2 = svm_loss_vectorized(W_rand, X_train, y_train, reg)
t3 = time.time()
print('Naive -- run time:', t2 - t1, '(s)')
print('Vectorized -- run time:', t3 - t2, '(s)')
print('loss difference:', np.linalg.norm(l1 - l2))
print('gradient difference:', np.linalg.norm(dW1 - dW2))

```

Kết quả:

```

Naive -- run time: 7.34640693665 (s)
Vectorized -- run time: 0.365024089813 (s)
loss difference: 8.73114913702e-11
gradient difference: 1.87942037251e-10

```

Kết quả cho thấy cách tính *vectorization* nhanh hơn so với cách tính *naive* khoảng 20 lần. Hơn nữa, sự chênh lệch giữa kết quả của hai cách tính là rất nhỏ, đều nhỏ hơn $1e-10$; ta có thể sử dụng cách tính *vectorization* để cập nhật nghiệm sử dụng mini-batch gradient descent.

29.3.3 Mini-batch gradient descent cho multi-class SVM

Với các hàm đã viết, chúng ta có thể thực hiện việc huấn luyện multi-class SVM bằng đoạn code dưới đây.

```
# Mini-batch gradient descent
def multiclass_svm_GD(X, y, Winit, reg, lr=.1, \
    batch_size = 1000, num_iters = 50, print_every = 10):
    W = Winit
    loss_history = []
    for it in xrange(num_iters):
        mix_ids = np.random.permutation(X.shape[0])
        n_batches = int(np.ceil(X.shape[0]/float(batch_size)))
        for ib in range(n_batches):
            ids = mix_ids[batch_size*ib: min(batch_size*(ib+1), X.shape[0])]
            X_batch = X[ids]
            y_batch = y[ids]
            lossib, dW = svm_loss_vectorized(W, X_batch, y_batch, reg)
            loss_history.append(lossib)
            W -= lr*dW
        if it % print_every == 0 and it > 0:
            print('it %d/%d, loss = %f' % (it, num_iters, loss_history[it]))
    return W, loss_history

d, C = X_train.shape[1], 10
reg = .1
W = 0.00001*np.random.randn(d, C)

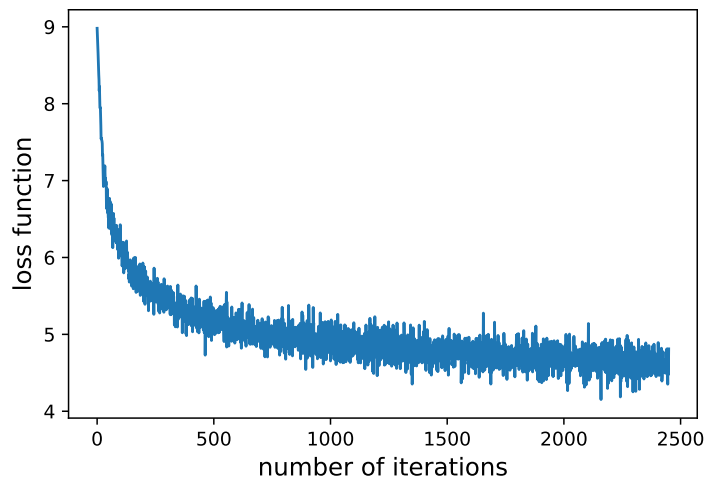
W, loss_history = multiclass_svm_GD(X_train, y_train, W, reg, lr = 1e-8, num_iters =
    50, print_every = 5)
```

Kết quả:

```
epoch 5/50, loss = 5.482782
epoch 10/50, loss = 5.204365
epoch 15/50, loss = 4.885159
epoch 20/50, loss = 5.051539
epoch 25/50, loss = 5.060423
epoch 30/50, loss = 4.691241
epoch 35/50, loss = 4.841132
epoch 40/50, loss = 4.643097
epoch 45/50, loss = 4.691177
```

Ta thấy rằng giá trị *loss* có xu hướng giảm và hội tụ. Giá trị này sau mỗi vòng lặp được minh họa trong Hình 29.6.

Sau khi đã tìm được ma trận hệ số \mathbf{W} đại diện cho mô hình multi-class SVM, chúng ta cần viết các hàm xác định nhãn của các điểm dữ liệu mới và đánh giá độ chính xác của mô hình như dưới đây:



Hình 29.6: Lịch sử *loss* qua các vòng lặp. Ta thấy rằng *loss* có xu hướng giảm và hội tụ khá nhanh.

```
def multisvm_predict(W, X):
    Z = X.dot(W)
    return np.argmax(Z, axis=1)

def evaluate(W, X, y):
    y_pred = multisvm_predict(W, X)
    acc = 100*np.mean(y_pred == y)
    return acc
```

Việc tiếp theo là sử dụng tập validation để chọn ra các bộ tham số mô hình phù hợp. Có hai tham số trong thuật toán tối ưu multi-class SVM: *regularization* và *learning rate*. Hai tham số này sẽ được tìm dựa trên các cặp giá trị cho trước. Bộ giá trị khiến cho độ chính xác của mô hình trên tập validation cao nhất sẽ được dùng để đánh giá tập kiểm thử.

```
lrs = [1e-9, 1e-8, 1e-7, 1e-6]
regs = [0.1, 0.01, 0.001, 0.0001]
best_W = 0
best_acc = 0
for lr in lrs:
    for reg in regs:
        W, loss_history = multiclass_svm_GD(X_train, y_train, W, reg, \
                                           lr = 1e-8, num_iters = 100, print_every = 1e20)
        acc = evaluate(W, X_val, y_val)
        print('lr = %e, reg = %e, loss = %f, validation acc = %.2f' % (lr, reg,
        loss_history[-1], acc))
        if acc > best_acc:
            best_acc = acc
            best_W = W
```


Kết quả:

```
lr = 1.000000e-09, reg = 1.000000e-01, loss = 4.422479, validation acc = 40.30
lr = 1.000000e-09, reg = 1.000000e-02, loss = 4.474095, validation acc = 40.70
lr = 1.000000e-09, reg = 1.000000e-03, loss = 4.240144, validation acc = 40.90
lr = 1.000000e-09, reg = 1.000000e-04, loss = 4.257436, validation acc = 41.40
lr = 1.000000e-08, reg = 1.000000e-01, loss = 4.482856, validation acc = 41.50
lr = 1.000000e-08, reg = 1.000000e-02, loss = 4.036566, validation acc = 41.40
lr = 1.000000e-08, reg = 1.000000e-03, loss = 4.085053, validation acc = 41.00
lr = 1.000000e-08, reg = 1.000000e-04, loss = 3.891934, validation acc = 41.40
lr = 1.000000e-07, reg = 1.000000e-01, loss = 3.947408, validation acc = 41.50
lr = 1.000000e-07, reg = 1.000000e-02, loss = 4.088984, validation acc = 41.90
lr = 1.000000e-07, reg = 1.000000e-03, loss = 4.073365, validation acc = 41.70
lr = 1.000000e-07, reg = 1.000000e-04, loss = 4.006863, validation acc = 41.80
lr = 1.000000e-06, reg = 1.000000e-01, loss = 3.851727, validation acc = 41.90
lr = 1.000000e-06, reg = 1.000000e-02, loss = 3.941015, validation acc = 41.80
lr = 1.000000e-06, reg = 1.000000e-03, loss = 3.995598, validation acc = 41.60
lr = 1.000000e-06, reg = 1.000000e-04, loss = 3.857822, validation acc = 41.80
```

Như vậy, độ chính xác cao nhất cho tập validation là 41.9%. Ma trận hệ số \mathbf{W} tốt nhất đã được lưu trong biến `best_W`. Áp dụng mô hình này lên tập kiểm thử:

```
acc = evaluate(best_W, X_test, y_test)
print('Accuracy on test data = %2f %%'%acc)
```

Kết quả:

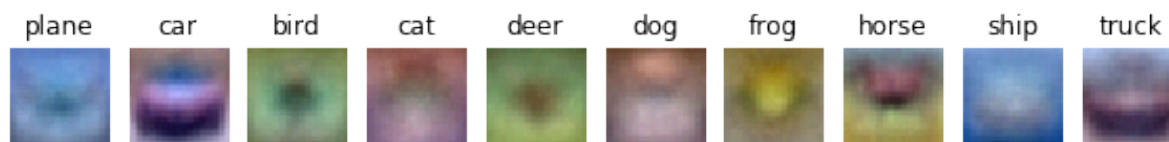
```
Accuracy on test data = 39.88 %
```

Như vậy, kết quả đạt được rơi vào khoảng gần 40 %. Bạn đọc có thể thử với các bộ tham số khác và có thể đạt được kết quả tốt hơn một vài phần trăm.

29.3.4 Minh họa nghiệm tìm được

Để ý rằng mỗi \mathbf{w}_i có chiều giống như chiều của dữ liệu. Bằng cách bỏ ra các hệ số tương ứng với bias và sắp xếp lại các điểm của mỗi trong 10 vector hệ số tìm được, chúng ta sẽ thu được các *bức ảnh* cũng có kích thước $3 \times 32 \times 32$ như mỗi ảnh nhỏ trong cơ sở dữ liệu. Hình 29.7 mô tả hệ số tìm được của mỗi \mathbf{w}_i .

Ta thấy rằng hệ số tương ứng với mỗi lớp mô tả hình dạng khá giống với các bức ảnh trong lớp tương ứng, ví dụ như *car* và *truck* trông khá giống với các bức ảnh trong lớp *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 một con ngựa hai 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 lớp. Vì sao chúng ta có thể nói như vậy?



Hình 29.7: Minh họa hệ số tìm được dưới dạng các bức ảnh.

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

$$\text{class}(\mathbf{x}) = \arg \max_{i=1,2,\dots,C} \mathbf{w}_i^T \mathbf{x}$$

Để ý rằng 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 nhãn của một bức ảnh mới chính là việc đi tìm bức ảnh đó gần với bức ảnh *đại diện* cho lớp nào nhất. Việc này khá giống với 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. Lập luận này cũng được áp dụng với softmax regression.

29.4 Thảo luận

- Giống như softmax regression, multi-class SVM vẫn được coi là một bộ phân lớp tuyến tính vì đường ranh giới giữa các lớp 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 vì kỹ thuật được sử dụng vẫn là one-vs-rest. Một ưu điểm nữa của multi-class SVM là nó có thể được tối ưu bằng các phương pháp gradient descent, phù hợp với các bài toán với dữ liệu lớn. Việc đường ranh giới giữa các lớp là tuyến tính có thể được giải quyết bằng cách kết hợp nó với các deep neural network.
- 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 (xem <https://goo.gl/xLccj3>). 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, ta có thể thử cả hai phương pháp rồi chọn phương pháp cho kết quả tốt hơn.

Phương pháp nhân tử Lagrange

Việc tối thiểu (tối đa) một hàm số một biến liên tục, khả vi, với tập xác định là một tập mở¹ thường được thực hiện dựa trên việc giải phương trình đạo hàm của hàm số đó. Gọi hàm số đó là $f(x) : \mathbb{R} \rightarrow \mathbb{R}$, giá trị nhỏ nhất hoặc lớn nhất nếu có của nó thường được tìm bằng cách giải phương trình $f'(x) = 0$. Chú ý rằng điều ngược lại không đúng, tức một điểm thoả mãn đạo hàm bằng không chưa chắc đã làm cho hàm số đạt giá trị nhỏ nhất hoặc lớn nhất. 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 không.

Cách làm trên đây được áp dụng vào các bài toán tối ưu không ràng buộc, tức không có điều kiện nào của biến \mathbf{X} . Với bài toán mà ràng buộc là một phương trình:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{thoả mãn: } f_1(\mathbf{x}) &= 0 \end{aligned} \tag{A.1}$$

ta cũng có một phương pháp để đưa nó về bài toán không ràng buộc. Phương pháp này có tên là phương pháp nhân tử Lagrange.

Xét hàm số $\mathcal{L}(\mathbf{x}, \lambda) = f_0(\mathbf{x}) + \lambda f_1(\mathbf{x})$ với biến λ 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 (A.1) thoả mãn điều kiện $\nabla_{\mathbf{x}, \lambda} \mathcal{L}(\mathbf{x}, \lambda) = \mathbf{0}$. Điều này tương đương với:

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \lambda) = \nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \tag{A.2}$$

$$\nabla_{\lambda} \mathcal{L}(\mathbf{x}, \lambda) = f_1(\mathbf{x}) = 0 \tag{A.3}$$

Để ý rằng điều kiện thứ hai chính là ràng buộc trong bài toán (A.1).

¹ Xem thêm: *Open sets, closed sets and sequences of real numbers* (<https://goo.gl/AgKhCn>).

Việc giải hệ phương trình (A.2) - (A.3), trong nhiều trường hợp, đơn giản hơn việc trực tiếp đi tìm nghiệm của bài toán (A.1).

Ví dụ 1:

Tìm giá trị lớn nhất và nhỏ nhất của hàm số $f_0(x, y) = x + y$, biết rằng x, y thoả mãn điều kiện $f_1(x, y) = x^2 + y^2 = 2$.

Lời giải: Điều kiện ràng buộc có thể được viết lại dưới dạng $x^2 + y^2 - 2 = 0$. *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

$$\nabla_{x,y,\lambda} \mathcal{L}(x, y, \lambda) = 0 \Leftrightarrow \begin{cases} 1 + 2\lambda x = 0 \\ 1 + 2\lambda y = 0 \\ x^2 + y^2 = 2 \end{cases} \quad (\text{A.4})$$

Từ hai phương trình đầu của (A.4) ta suy ra $x = y = \frac{-1}{2\lambda}$. Thay vào phương trình cuối 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 bài toán.

Ví dụ 2: ℓ_2 norm của ma trận Chúng ta đã quen thuộc với ℓ_2 norm của một vector \mathbf{x} : $\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^T \mathbf{x}}$. Dựa trên ℓ_2 norm của vector, ℓ_2 norm của một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ được ký hiệu là $\|\mathbf{A}\|_2$ và được định nghĩa như sau:

$$\|\mathbf{A}\|_2 = \max \frac{\|\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \max \sqrt{\frac{\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x}}{\mathbf{x}^T \mathbf{x}}}, \text{ với } \mathbf{x} \in \mathbb{R}^n \quad (\text{A.5})$$

Bài toán tối ưu này tương đương với:

$$\begin{aligned} & \max (\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x}) \\ & \text{thoả mãn: } \mathbf{x}^T \mathbf{x} = 1 \end{aligned} \quad (\text{A.6})$$

Lagrangian của bài toán này là

$$\mathcal{L}(\mathbf{x}, \lambda) = \mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} + \lambda(1 - \mathbf{x}^T \mathbf{x}) \quad (\text{A.7})$$

Các điểm cực trị của hàm số Lagrange phải thoả mãn

$$\nabla_{\mathbf{x}} \mathcal{L} = 2\mathbf{A}^T \mathbf{A} \mathbf{x} - 2\lambda \mathbf{x} = \mathbf{0} \quad (\text{A.8})$$

$$\nabla_{\lambda} \mathcal{L} = 1 - \mathbf{x}^T \mathbf{x} = 0 \quad (\text{A.9})$$

Từ (A.8) ta có $\mathbf{A}^T \mathbf{A} \mathbf{x} = \lambda \mathbf{x}$. Vậy \mathbf{x} phải là một vector riêng của $\mathbf{A}^T \mathbf{A}$ và λ chính là trị riêng tương ứng. Nhân cả hai vế của biểu thức này với \mathbf{x}^T vào bên trái và sử dụng (A.9), ta thu được

$$\mathbf{x}^T \mathbf{A}^T \mathbf{A} \mathbf{x} = \lambda \mathbf{x}^T \mathbf{x} = \lambda \quad (\text{A.10})$$

Từ đó suy ra $\|\mathbf{A}\mathbf{x}\|_2$ đạt giá trị lớn nhất khi λ đạt giá trị lớn nhất. Nói cách khác, λ phải là trị riêng lớn nhất của $\mathbf{A}^T \mathbf{A}$. Vậy, $\|\mathbf{A}\|_2 = \lambda_{\max}(\mathbf{A}^T \mathbf{A})$.

Các trị riêng của $\mathbf{A}^T \mathbf{A}$ còn được gọi là *singular value* của \mathbf{A} . Tóm lại, ℓ_2 norm của một ma trận là singular value lớn nhất của ma trận đó.

Hoàn toàn tương tự, nghiệm của bài toán

$$\min_{\|\mathbf{x}\| \leq 1} \|\mathbf{A}\mathbf{x}\|_2 \tag{A.11}$$

chính là một vector riêng ứng với singular value nhỏ nhất của \mathbf{A} .

Tài liệu tham khảo

- AKA91. David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.
- AM93. Sunil Arya and David M Mount. Algorithms for fast vector quantization. In *Data Compression Conference, 1993. DCC'93.*, pages 381–390. IEEE, 1993.
- AMMIL12. Yaser S Abu-Mostafa, Malik Magdon-Ismael, and Hsuan-Tien Lin. *Learning from data*, volume 4. AML-Book New York, NY, USA:, 2012.
- AV07. David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.
- Bis06. Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
- BL14. Artem Babenko and Victor Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 931–938, 2014.
- Ble08. David M Blei. Hierarchical clustering. 2008.
- BMV⁺12. Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.
- BTVG06. Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.
- BV04. Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.
- CDF⁺04. Gabriella Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. Visual categorization with bags of keypoints. In *Workshop on statistical learning in computer vision, ECCV*, volume 1, pages 1–2. Prague, 2004.
- CLMW11. Emmanuel J Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):11, 2011.
- Cyb89. George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.
- DFK⁺04. Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V Vinay. Clustering large graphs via the singular value decomposition. *Machine learning*, 56(1):9–33, 2004.
- dGJL05. Alexandre d’Aspremont, Laurent E Ghaoui, Michael I Jordan, and Gert R Lanckriet. A direct formulation for sparse pca using semidefinite programming. In *Advances in neural information processing systems*, pages 41–48, 2005.
- DHS11. John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.
- DT05. Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.
- ERK⁺11. Michael D Ekstrand, John T Riedl, Joseph A Konstan, et al. Collaborative filtering recommender systems. *Foundations and Trends® in Human-Computer Interaction*, 4(2):81–173, 2011.
- FHT01. Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.
- Fuk13. Keinosuke Fukunaga. *Introduction to statistical pattern recognition*. Academic press, 2013.
- GBC16. Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.

- GR70. Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Numerische mathematik*, 14(5):403–420, 1970.
- HNO06. Per Christian Hansen, James G Nagy, and Dianne P O’leary. *Deblurring images: matrices, spectra, and filtering*. SIAM, 2006.
- HZRS16. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- JDJ17. Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.
- JDS11. Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.
- KA04. Shehroz S Khan and Amir Ahmad. Cluster center initialization algorithm for k-means clustering. *Pattern recognition letters*, 25(11):1293–1302, 2004.
- KB14. Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
- KBV09. Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8), 2009.
- KH92. Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.
- KSH12. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- LCB10. Yann LeCun, Corinna Cortes, and Christopher JC Burges. Mnist handwritten digit database. *AT&T Labs [Online]*. Available: <http://yann.lecun.com/exdb/mnist>, 2, 2010.
- LCD04. Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 219–230. ACM, 2004.
- Low99. David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.
- LSP06. Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, volume 2, pages 2169–2178, 2006.
- LW⁺02. Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.
- M⁺97. Tom M Mitchell et al. Machine learning. wcb, 1997.
- MSS⁺99. Sebastian Mika, Bernhard Schölkopf, Alex J Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. In *Advances in neural information processing systems*, pages 536–542, 1999.
- Nes07. Yurii Nesterov. Gradient methods for minimizing composite objective function, 2007.
- NF13. Mohammad Norouzi and David J Fleet. Cartesian k-means. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3017–3024, 2013.
- NJW02. Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002.
- Pat07. Arkadiusz Paterek. Improving regularized singular value decomposition for collaborative filtering. In *Proceedings of KDD cup and workshop*, volume 2007, pages 5–8, 2007.
- Pla98. John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.
- Pri12. Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.
- RDVC⁺04. Lorenzo Rosasco, Ernesto De Vito, Andrea Caponnetto, Michele Piana, and Alessandro Verri. Are loss functions all the same? *Neural Computation*, 16(5):1063–1076, 2004.
- Rey15. Douglas Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, pages 827–832, 2015.
- Ros57. F Rosembat. The perceptron: A perceiving and recognizing automation. *Cornell Aeronautical Laboratory Report*, 1957.
- Rud16. Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- SCSC03. Mei-Ling Shyu, Shu-Ching Chen, Kanoksri Sarinnapakorn, and LiWu Chang. A novel anomaly detection scheme based on principal component classifier. Technical report, MIAMI UNIV CORAL GABLES FL DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 2003.
- SFHS07. J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.
- SHK⁺14. Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

- SKKR00. Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Application of dimensionality reduction in recommender system-a case study. Technical report, Minnesota Univ Minneapolis Dept of Computer Science, 2000.
- SKKR02. Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth International Conference on Computer and Information Science*, pages 27–28. Citeseer, 2002.
- SLJ⁺15. Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.
- SSWB00. Bernhard Schölkopf, Alex J Smola, Robert C Williamson, and Peter L Bartlett. New support vector algorithms. *Neural computation*, 12(5):1207–1245, 2000.
- SWY75. Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
- SZ14. Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- TH12. Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.
- VJG14. João Vinagre, Alípio Mário Jorge, and João Gama. Fast incremental matrix factorization for recommendation with positive-only feedback. In *International Conference on User Modeling, Adaptation, and Personalization*, pages 459–470. Springer, 2014.
- VL07. Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.
- VM16. Tiep Vu and Vishal Monga. Learning a low-rank shared dictionary for object classification. In *Proceedings IEEE Int. Conference on Image Processing*, pages 4428–4432. IEEE, 2016.
- VM17. Tiep Vu and Vishal Monga. Fast low-rank shared dictionary learning for image classification. *IEEE Transactions on Image Processing*, 26(11):5160–5175, Nov 2017.
- VMM⁺16. Tiep Vu, Hojjat Seyed Mousavi, Vishal Monga, Ganesh Rao, and UK Arvind Rao. Histopathological image classification using discriminative feature-oriented dictionary learning. *IEEE transactions on medical imaging*, 35(3):738–751, 2016.
- WYG⁺09. John Wright, Allen Y Yang, Arvind Ganesh, S Shankar Sastry, and Yi Ma. Robust face recognition via sparse representation. *IEEE transactions on pattern analysis and machine intelligence*, 31(2):210–227, 2009.
- XWCL15. Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.
- YZFZ11. M. Yang, L. Zhang, X. Feng, and D. Zhang. Fisher discrimination dictionary learning for sparse representation. pages 543–550, Nov. 2011.
- ZDW14. Ting Zhang, Chao Du, and Jingdong Wang. Composite quantization for approximate nearest neighbor search. In *ICML*, number 2, pages 838–846, 2014.
- ZF14. Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.
- ZWFM06. Sheng Zhang, Weihong Wang, James Ford, and Fillia Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pages 549–553. SIAM, 2006.
- ZYK06. Haitao Zhao, Pong Chi Yuen, and James T Kwok. A novel incremental principal component analysis and its application for face recognition. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 36(4):873–886, 2006.
- ZYX⁺08. Zhi-Qiang Zeng, Hong-Bin Yu, Hua-Rong Xu, Yan-Qi Xie, and Ji Gao. Fast training support vector machines using parallel sequential minimal optimization. In *Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on*, volume 1, pages 997–1001. IEEE, 2008.

Index

- α -sublevel sets, 294
- activation function
 - sigmoid function, 167
 - tanh function, 167
- activation function, 162
 - ReLU, 199
- activation function–hàm kích hoạt, 197
- affine function, 290

- back substitution, 16
- backpropagation, 200
- bag of words, 75
 - dictionary, 76
- basic, 19
 - orthogonal, 20
 - orthonormal, 20
- batch gradient descent, 152
- Bayes' rule - quy tắc Bayes, 44
- bias, 86
- bias trick, 86, 366
- binary classification, 156

- class boundary, 156
- classification–phân lớp, 65
- closed-form solution, 97
- cluster, 110
- complementary slackness, 323
- conditional probability - xác suất có điều kiện, 44
- conjugate distributions, 59
- conjugate prior, 59
- constraints, 282
- contours, 293
- convex, 282
 - combination, 287
 - function, 288
 - domain, 288
 - functions
 - first-order condition, 296
 - Second-order condition, 297
 - hull, 287
 - optimization problems, 305
 - sets, 283
 - strictly convex functions, 289

- convex optimization, 282
- cosine similarity, 228
- cross entropy, 184
- CVXOPT, 307

- data point - điểm dữ liệu, 64
- determinant, 16
- diagonal matrix, 15
- dimensionality reduction, 75
- dimensionality reduction – giảm chiều dữ liệu, 245
- duality, 317

- early stopping, 96
- eigenvalues, 22
- eigenvectors, 22
- elbow method, 123
- element-wise, 197
- epoch, 153
- expectation - kỳ vọng, 45

- feasible points, 282
- feasible sets, 282
- feature engineering, 71
- feature extraction, 245
- feature selection, 97, 245
- feature vector, 71
- feature vector - vector đặc trưng, 64
- Fisher's linear discriminant, 273
- forward substitution, 16

- Gaussian naive Bayes, 128
- Gaussian mixture model, 124
- GD, *see* gradient descent
- generalization, 91
- Geometric Programming, 313
- Geometric programming
 - convex form, 315
- global minimum, 140
- gradient descent, 140
 - stopping criteria – điều kiện dừng, 155
 - batch size, 154
 - momentum, 148
 - Nesterov accelerated gradient, 151
- gradient–đạo hàm, 30

- first-order gradient–đạo hàm bậc nhất, 30
- numerical gradient, 36
- second-order gradient–đạo hàm bậc hai, 30
- ground truth, 83
- Hadamard product, 202, 203
- halfspace, 285
- hand-crafted feature, 79
- Hermitian, 13
- hidden layer, 162
- hierarchical, 176
- hierarchical clustering, 120
- hinge loss, 346
- hinge loss tổng, 368
- Huber loss, 89
- hyperparameter, 60
- hyperplane, 156
- hyperplane – siêu mặt phẳng, 285
- hyperpolygon–siêu đa diện, 111
- identity matrix - ma trận đơn vị, 14
- infeasible sets, 282
- inner product – tích vô hướng, 14
- input layer, 162
- inverse matrix - ma trận nghịch đảo, 15
- iteration, 153
- joint probability - xác suất đồng thời, 41
- K -means clustering, 110
- K -nearest neighbor, 100
- Kernel, 358
 - Kernel trick, 358
 - Linear, 359
 - Mercer conditions, 359
 - Polynomial, 360
 - Radial Basic Function (RBF), 360
 - Sigmoid, 360
- KKT conditions, 324
- KNN, *xem* K -nearest neighbor, 100
- Lagrange
 - dual function, 318
 - dual problem, 321
 - Lagrangian, 318
- Lagrange/Lagrangian
 - dual functions, 318
- Laplace smoothing, 129
- large-scale, 101
- lasso regression, 97
- lazy learning, 100
- LDA, 269
- learning rate, 141
- lemmatization, 133
- level sets, 293
- level sets–đường đồng mức, 147
- likelihood, 53
- linear combination, 17
- linear dependence, 17
- linear discriminant analysis, 269
- linear independence, 17
- linear programming, 307
 - general form, 308
 - standard form, 308
- linear regression–hồi quy tuyến tính, 83
- linearly separable, 156
- Ling-Spam dataset, 132
- local minimum, 140
- log-likelihood, 53
- loss function–hàm mất mát, 69
- MAP, 58
- marginal probability - xác suất biên, 43
- marginalization, 43
- matrix calculus, 30
- matrix completion, 216
- matrix factorization: phân tích ma trận thành nhân tử, 236
- maximum a posteriori, 58
- maximum entropy classifier, 191
- maximum likelihood estimation, 53
- maximum margin classifier, 330
- mean squared error, 93
- mini-batch gradient descent, 154
- misclassified point–điểm bị phân lớp lỗi, 158
- MLE, 53
- MNIST, 117
- model parameter–tham số mô hình, 69
- model parameters, 69
- monomial, 313
- multi-class classification, 175
- multinomial logistic regression, 191
- multinomial naive Bayes, 129
- naive Bayes classifier, 127
- NBC, 127
- neural network, 162
- non-word, 133
- norm, 26
 - ℓ_1 norm, 27
 - ℓ_2 norm, 27
 - ℓ_p norm, 27
 - Euclidean norm, 27
 - Frobenius norm, 28
- norm balls, 285
- null space, 19
- numpy, *iv*
- offline learning, 67
- one-hot coding, 111
- one-vs-one, 176
- one-vs-rest, 177
- online learning, 67, 152
- orthogonal matrix, 20
- orthogonality, 20
- output layer, 162
- overfitting, 91

- partial derivative–đạo hàm riêng, 30
- patch, 77
- PCA–*xem* principle component analysis, 254
- pdf, *xem* probability density function, 40
- perceptron learning algorithm, 156
- PLA, 156
- pocket algorithm, 163
- polynomial regression, 89, 92
- positive definite matrix, 24
 - negative definite, 24
 - negative semidefinite, 24
 - positive semidefinite, 24
- posterior probability, 58
- posynomial, 313
- predicted output, 83
- principal component analysis, 254
- prior, 58
- probability density function - hàm mật độ xác suất, 40
- probability distribution - phân phối xác suất, 47
 - Bernoulli distribution, 47
 - Beta distribution, 50
 - Categorical distribution, 48
 - Dirichlet distribution, 51
 - multivariate normal distribution, 50
 - univariate normal distribution, 49
- projection matrix, 75, 269
- pseudo inverse, 85

- quadratic
 - forms, 291
- Quadratic programming, 310
- quasiconvex, 296

- random projection, 75
- random variable - biến ngẫu nhiên, 40
- range space, 19
- rank, 19
- recommendation system
 - collaborative filtering, 215
 - content-based, 214, 215
 - item, 214
 - item-item collaborative filtering, 230
 - long tail, 214
 - similarity matrix, 228
 - user, 214
 - user-user collaborative filtering, 226
 - utility matrix, 215
- regression–hồi quy, 65
- regularization, 96
 - ℓ_1 regularization, 97
 - ℓ_2 regularization, 97
- regularization parameter, 97
- regularized loss function, 97
- regularized neural network, 209
- reinforcement learning - học củng cố, 68
- ridge regression, 90, 97
- robust, 97

- scikit-learn, iv
- semi-supervised learning–học bán giám sát, 68
- Separating hyperplane theorem, 288
- SGD, *see* stochastic gradient descent
- sigmoid, 198
- sklearn, iv
- Slater’s constraint qualification, 322
- softmax function, 181
- softmax regression, 180
- spam filtering, 132
- span, 17
- sparsity, 97
- spectral clustering, 124
- state-of-the-art, 74
- stochastic gradient descent, 152
- stop word, 133
- strong duality, 322
- submatrix
 - leading principal matrix, 25
 - leading principal minor, 25
 - principal minor, 25
 - principal submatrix, 25
- supervised learning–học có giám sát, 67
- Support Vector Machine, 328
 - Hard Margin SVM, 328
- Support vector machine
 - Kernel SVM, 355
 - soft-margin SVM, 339
- support vector machine
 - margin, 329
 - multi-class SVM, 364
- symmetric matrix, 13

- tanh, 198
- task, 64
- tensor, 64
- test set - tập kiểm thử, 67
- training error, 93
- training set - tập huấn luyện, 67
- transfer learning, 80
- triangular matrix, 16
 - lower, 16
 - upper, 16

- underfitting, 92
- unitary matrix, 21
- unsupervised learning–học không giám sát, 68

- validation, 94
 - cross-validation, 95
 - k-fold cross-validation, 95
 - leave-one-out, 95
- vector-valued function, 31
- vectorization–vector hoá, 74

- weak duality, 321
- weight decay, 97
- weight vector–vector trọng số, 83