Vũ Hữu Tiệp

Machine Learning cơ bản

Theo blog: http://machinelearningcoban.com

(*Đang trong quá trình xây dựng*)

First Edition

March 22, 2017

Contents

1

Giới thiệu

Những năm gần đây, AI - Artificial Intelligence (Trí Tuệ Nhân Tạo), và cụ thể hơn là Machine Learning (Học Máy hoặc Máy Học - *Với những từ chuyên ngành, tôi sẽ dùng song song cả tiếng Anh và tiếng Việt, tuy nhiên sẽ ưu tiên tiếng Anh vì thuận tiện hơn trong việc tra cứu*) 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 đang len lỏi vào mọi 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, máy chơi cờ vây AlphaGo của Google DeepMind, ..., chỉ là một vài trong vô vàn những ứng dụng của AI/Machine Learning. (Xem thêm Jarvis - trợ lý thông minh cho căn nhà của Mark Zuckerberg.)

Machine Learning là một tập con của AI. Theo định nghĩa của Wikipedia, *Machine learning is the subfield of computer science that "gives computers the ability to learn without being explicitly programmed"*. Nói đơn giản, Machine Learning là một lĩnh vực nhỏ của Khoa Học Máy Tính, nó có khả năng tự học hỏi dựa trên dữ liệu đưa vào mà không cần phải được lập trình cụ thể. Bạn Nguyễn Xuân Khánh tại đại học Maryland đang viết một cuốn sách về Machine Learning bằng tiếng Việt khá thú vị, các bạn có thể tham khảo bài Machine Learning là gì?.

Những năm gần đây, khi mà khả năng tính toán của các máy tính được nâng lên một tầm cao mới và lượng dữ liệu khổng lồ được thu thập bởi các hãng công nghệ lớn, Machine Learning đã tiến thêm một bước tiến dài và một lĩnh vực mới được ra đời gọi là Deep Learning (Học Sâu). Deep Learning đã giúp máy tính thực thi những việc tưởng chừng như không thể vào 10 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, hay thậm chí cả sáng tác văn hay âm nhạc (Xem thêm 8 Inspirational Applications of Deep Learning).

Mối quan hệ giữa Artificial Intelligence, Machine Learning, và Deep Learning được cho trong Hình 1.1.

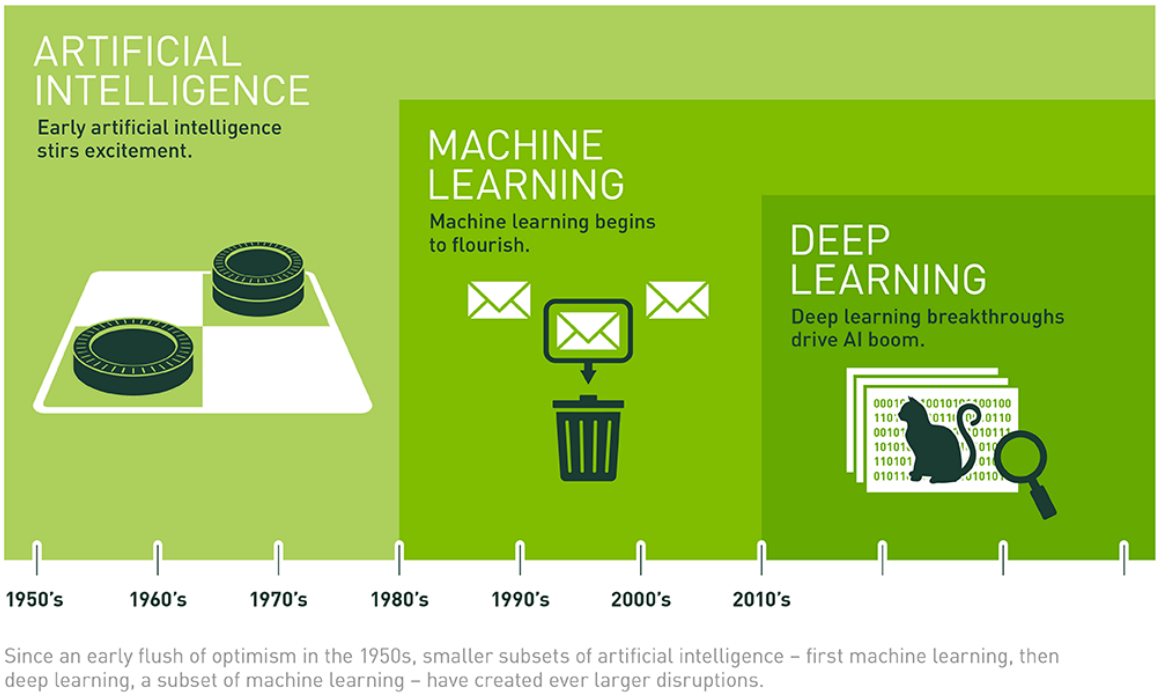

Fig. 1.1: Mối quan hệ giữa AI, Machine Learning và Deep Learning. Nguồn: What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?

1.1 Mục đích viết Blog

Nhu cầu về nhân lực ngành Machine Learning (Deep Learning) đang ngày một cao, kéo theo đó nhu cầu học Machine Learning trên thế giới và ở Việt Nam ngày một lớn. Cá nhân tôi cũng muốn hệ thống lại kiến thức của mình về lĩnh vực này để chuẩn bị cho tương lai (đây là một trong những mục tiêu của tôi trong năm 2017). Tôi sẽ cố gắng đi từ những thuật toán cơ bản nhất của Machine Learning kèm theo các ví dụ và mã nguồn trong mỗi bài viết. Tôi sẽ viết 1-2 tuần 1 bài (việc viết các công thức toán và code trên blog thực sự tốn nhiều thời gian hơn tôi từng nghĩ). Đồng thơi, tôi cũng mong muốn nhận được phản hồi của bạn đọc để qua những thảo luận, tôi và các bạn có thể nắm bắt được các thuật toán này.

Khi chuẩn bị các bài viết, tôi sẽ giả định rằng bạn đọc có một chút kiến thức về Đại Số Tuyến Tính (Linear Algebra), Xác Suất Thống Kê (Probability and Statistics) và có kinh nghiệm về lập trình Python. Nếu bạn chưa có nhiều kinh nghiệm về các lĩnh vực này, đừng quá lo lắng vì mỗi bài sẽ chỉ sử dụng một vài kỹ thuật cơ bản. Hãy để lại câu hỏi của bạn ở phần Comment bên dưới mỗi bài, tôi sẽ thảo luận thêm với các bạn.

Trong bài tiếp theo của blog này, tôi sẽ giới thiệu về các nhóm thuật toán Machine learning cơ bản. Mời các bạn theo dõi.

1.2 Tham khảo thêm

1.2.1 Các khóa học

Tiếng Anh

1. Machine Learning với thầy Andrew Ng trên Coursera (*Khóa học nổi tiếng nhất về Machine Learning*)
2. Deep Learning by Google trên Udacity (*Khóa học nâng cao hơn về Deep Learning với Tensorflow*)
3. Machine Learning mastery (*Các thuật toán Machine Learning cơ bản*)

Tiếng Việt

Lưu ý: *Các khóa học này tôi chưa từng tham gia, chỉ đưa ra để các bạn tham khảo.*

1. Machine Learning 1/2017
2. Nhập môn Machine Learning, Tech Master- Cao Thanh Hà *POSTECH*()

1.2.2 Các trang Machine Learning tiếng Việt khác

1. Machine Learning trong Xử Lý Ngôn Ngữ Tự Nhiên - Nhóm Đông Du *Nhật Bản*
2. Machine Learning cho người mới bắt đầu - Ông Xuân Hồng *JAIST*.
3. Machine Learning book for Vietnamese - Nguyễn Xuân Khánh *University of Maryland*

2

Phân nhóm các thuật toán Machine Learning

Có hai cách phổ biến phân nhóm các thuật toán Machine learning. Một là dựa trên phương thức học (learning style), hai là dựa trên chức năng (function) (của mỗi thuật toán).

2.1 Phân nhóm dựa trên phương thức học

Theo phương thức học, các thuật toán Machine Learning thường được chia làm 4 nhóm: Supervise learning, Unsupervised learning, Semi-supervised lerning và Reinforcement learning. *Có một số cách phân nhóm không có Semi-supervised learning hoặc Reinforcement learning.*

2.1.1 Supervised Learning (Học có giám sát)

Supervised learning là thuật toán dự đoán đầu ra (outcome) của một dữ liệu mới (new input) dựa trên các cặp (*input, outcome*) đã biết từ trước. Cặp dữ liệu này còn được gọi là (*data. label*), tức (*dữ liệu, nhãn*). Supervised learning là nhóm phổ biến nhất trong các thuật toán Machine Learning.

Một cách toán học, Supervised learning là khi chúng ra có một tập hợp biến đầu vào $\mathcal{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ và một tập hợp nhãn tương ứng $\mathcal{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N\}$, trong đó $\mathbf{x}_i, \mathbf{y}_i$ là các vector. Các cặp dữ liệu biết trước $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y}$ được gọi là tập *training data* (dữ liệu huấn luyện). Từ tập traing data 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 $\mathcal{Y}$:

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

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 nhãn tương ứng của nó $\mathbf{y} = f(\mathbf{x})$.

Ví dụ 1: trong nhận dạng chữ viết tay (Hình 2.1), ta có ảnh của hàng nghìn ví dụ của mỗi chữ số được viết bởi nhiều người khác nhau. Chúng ta đưa các bức ảnh này vào trong một

thuật toán và chỉ cho nó biết mỗi bức ảnh tương ứng với chữ số nào. Sau khi thuật toán tạo ra (sau khi *học*) 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.

Fig. 2.1: MNIST: bộ cơ sở dữ liệu của chữ số viết tay. (Nguồn: Simple Neural Network implementation in Ruby)

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 trong một cuốn sách mà chúng chưa nhìn thấy bao giờ.

Ví dụ 2: Thuật toán dò các khuôn mặt trong một bức ảnh đã được phát triển từ rất lâu. Thời gian đầu, facebook sử dụng thuật toán này để chỉ ra các khuôn mặt trong một bức ảnh và yêu cầu người dùng *tag friends* - tức gán nhãn cho mỗi khuôn mặt. Số lượng cặp dữ liệu (*khuôn mặt, tên người*) càng lớn, độ chính xác ở những lần tự động *tag* tiếp theo sẽ càng lớn.

Ví dụ 3: Bản thân thuật toán dò tìm các khuôn mặt trong 1 bức ảnh cũng là một thuật toán Supervised learning với training data (dữ liệu học) là hàng ngàn cặp (*ảnh, mặt người*) và (*ảnh, không phải mặt người*) được đưa vào. Chú ý là dữ liệu này chỉ phân biệt *mặt người* và *không phải mặt ngươi* mà không phân biệt khuôn mặt của những người khác nhau.

Thuật toán supervised learning còn được tiếp tục chia nhỏ ra thành hai loại chính:

Classification (Phân loại)

Một bài toán được gọi là *classification* nếu các *label* của *input data* được chia thành một số hữu hạn nhóm. Ví dụ: Gmail xác định xem một email có phải là spam hay không; các hãng tín dụng xác định xem một khách hàng có khả năng thanh toán nợ hay không. Ba ví dụ phía trên được chia vào loại này.

Regression (Hồi quy)

(tiếng Việt dịch là *Hồi quy*, tôi không thích cách dịch này vì bản thân không hiểu nó nghĩa là gì)

Nếu *label* không được chia thành các nhóm mà là một giá trị thực cụ thể. Ví dụ: một căn nhà rộng x m^2, có y phòng ngủ và cách trung tâm thành phố z km sẽ có giá là bao nhiêu?

Gần đây Microsoft có một ứng dụng dự đoán giới tính và tuổi dựa trên khuôn mặt. Phần dự đoán giới tính có thể coi là thuật toán **Classification**, phần dự đoán tuổi có thể coi là 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.*

2.1.2 Unsupervised Learning (Học không giám sát)

Trong thuật toán này, chúng ta không biết được *outcome* hay *nhãn* mà chỉ có dữ liệu đầu vào. 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 (clustering) hoặc giảm số chiều của dữ liệu (dimention reduction) để thuận tiện trong việc lưu trữ và tính toán.

Một cách toán học, Unsupervised learning là khi chúng ta chỉ có dữ liệu vào $\mathcal{X}$ mà không biết *nhãn* $\mathcal{Y}$ tương ứng.

Những thuật toán loại này được gọi là Unsupervised learning vì không giống như Supervised learning, chúng ta không biết câu trả lời chính xác cho mỗi dữ liệu đầu vào. Giống như khi ta học, không có thầy cô giáo nào chỉ cho ta biết đó là chữ A hay chữ B. Cụm *không giám sát* được đặt tên theo nghĩa này.

Các bài toán Unsupervised learning được tiếp tục chia nhỏ thành hai loại:

Clustering (phân nhóm)

Một 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.

Association

Là bài toán khi chúng ta muốn khám phá ra một quy luật dựa trên nhiều dữ liệu cho trước. Ví dụ: những khách hàng nam mua quần áo thường có xu hướng mua thêm đồng hồ hoặc thắt lưng; những khán giả xem phim Spider Man thường có xu hướng xem thêm phim Bat Man, dựa vào đó tạo ra một hệ thống gợi ý khách hàng (Recommendation System), thúc đẩy nhu cầu mua sắm.

2.1.3 Semi-Supervised Learning (Học bán giám sát)

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 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 toàn 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 thậm chí cần phải có chuyên gia mới gán nhãn được (ảnh y học chẳng hạn). Ngược lại, dữ liệu chưa có nhãn có thể được thu thập với chi phí thấp từ internet.

2.1.4 Reinforcement Learning (Học Củng Cố)

Reinforcement learning là các bài toán giúp cho một hệ thống tự động xác định hành vi dựa trên hoàn cảnh để đạt được lợi ích cao nhất (maximizing the performance). Hiện tại, Reinforcement learning chủ yếu được áp dụng vào Lý Thuyết Trò Chơi (Game Theory), các thuật toán cần xác định nước đi tiếp theo để đạt được điểm số cao nhất.

Ví dụ 1: AlphaGo (Hình 2.2) gần đây nổi tiếng với việc chơi cờ vây thắng cả con người. Cờ vây được xem là có độ phức tạp cực kỳ cao 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}!! Vì vậy, thuật toán phải chọn ra 1 nước đi tối ưu trong số hàng nhiều tỉ tỉ lựa chọn, và tất nhiên, không thể áp dụng thuật toán tương tự như IBM Deep Blue) (IBM Deep Blue đã thắng con người trong môn cờ vua 20 năm trước). 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. (Xem thêm tại Google DeepMind's AlphaGo: How it works).

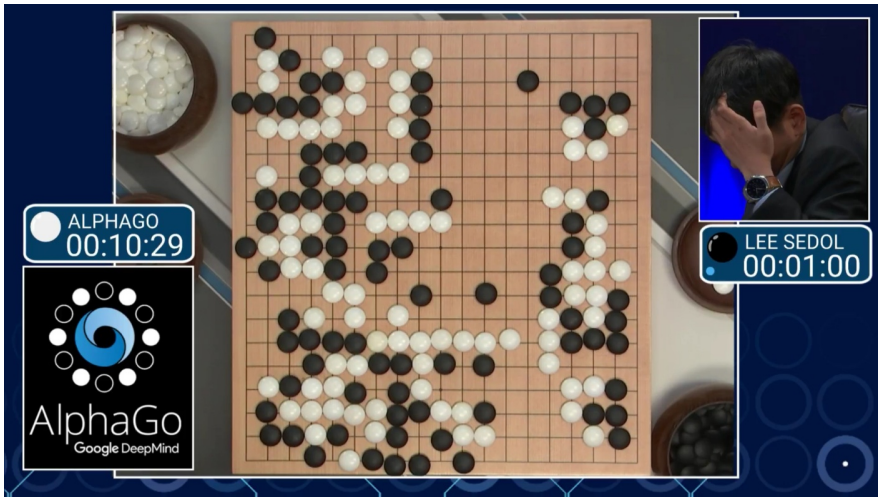

Fig. 2.2: AlphaGo chơi cờ vây với Lee Sedol. AlphaGo là một ví dụ của Reinforcement learning (Nguồn AlphaGo AI Defeats Sedol Again, With 'Near Perfect Game')

Ví dụ 2: Huấn luyện cho máy tính chơi game Mario. 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 ưu để tối đa số điểm trên, qua đó đạt được mục đích cuối cùng là cứu công chúa.

2.2 Phân nhóm dựa trên chức năng

Có một cách phân nhóm thứ hai dựa trên chức năng của các thuật toán. Trong phần này, tôi xin chỉ liệt kê các thuật toán. Thông tin cụ thể sẽ được trình bày trong các bài viết khác tại blog này. Trong quá trình viết, tôi có thể sẽ thêm bớt một số thuật toán.

2.2.1 Regression Algorithms

1. Linear Regression
2. Logistic Regression
3. Stepwise Regression

2.2.2 Classification Algorithms

1. Linear Classifier
2. Support Vector Machine (SVM)
3. Kernel SVM
4. Sparse Represntation-based classification (SRC)

2.2.3 Instance-based Algorithms

1. k-Nearest Neighbor (kNN)
2. Learnin Vector Quantization (LVQ)

2.2.4 Regularization Algorithms

1. Ridge Regression
2. Least Absolute Shringkage and Selection Operator (LASSO)
3. Least-Angle Regression (LARS)

2.2.5 Bayesian Algorithms

1. Naive Bayes
2. Gausian Naive Bayes

2.2.6 Clustering Algorithms

1. k-Means clustering
2. k-Medians
3. Expectation Maximisation (EM)

2.2.7 Artificial Neural Network Algorithms

1. Perceptron
2. Softmax Regression
3. Multi-layer Perceptron
4. Back-Propagation

2.2.8 Dimensionality Reduction Algorithms

1. Principal Component Analysis (PCA)
2. Linear Discriminant Analysis (LDA)

2.2.9 Ensemble Algorithms

1. Boosting
2. AdaBoost
3. Random Forest

Và còn rất nhiều các thuật toán khác.

2.3 Tài liệu tham khảo

1. A Tour of Machine Learning Algorithms
2. Điểm qua các thuật toán Machine Learning hiện đại

3

Linear Regression

Trong bài này, tôi sẽ giới thiệu một trong những thuật toán cơ bản nhất (và đơn giản nhất) của Machine Learning. Đây là một thuật toán *Supervised learning* có tên **Linear Regression** (Hồi Quy Tuyến Tính). Bài toán này đôi khi được gọi là *Linear Fitting* (trong thống kê) hoặc *Linear Least Square*.

3.1 Giới thiệu

Quay lại ví dụ đơn giản được nêu trong bài trước: một căn nhà rộng x_1 m^2, có x_2 phòng ngủ và cách trung tâm thành phố x_3 km có giá là bao nhiêu. Giả sử chúng ta đã có số liệu thống kê từ 1000 căn nhà trong thành phố đó, liệu rằng khi có một căn nhà mới với các thông số về diện tích, số phòng ngủ và khoảng cách tới trung tâm, chúng ta có thể dự đoán được giá của căn nhà đó không? Nếu có thì hàm dự đoán $y = f(\mathbf{x})$ sẽ có dạng như thế nào. Ở đây $\mathbf{x} = [x_1, x_2, x_3]$ là một vector hàng chứa thông tin *input*, y là một số vô hướng (scalar) biểu diễn *output* (tức giá của căn nhà trong ví dụ này).

Lưu ý về ký hiệu toán học: *trong các bài viết của tôi, các số vô hướng được biểu diễn bởi các chữ cái viết ở dạng không in đậm, có thể viết hoa, ví dụ x_1, N, y, k. Các vector được biểu diễn bằng các chữ cái thường in đậm, ví dụ $\mathbf{y}, \mathbf{x}_1$. Các ma trận được biểu diễn bởi các chữ viết hoa in đậm, ví dụ $\mathbf{X}, \mathbf{Y}, \mathbf{W}$.*

Một cách đơn giản nhất, chúng 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. Một hàm số đơn giản nhất có thể mô tả mối quan hệ giữa giá nhà và 3 đại lượng đầu vào là:

$$y \approx \hat{y} = f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_0 \tag{3.1}$$

trong đó, w_1, w_2, w_3, w_0 là các hằng số, w_0 còn được gọi là bias. Mối quan hệ $y \approx f(\mathbf{x})$ bên trên là một mối quan hệ tuyến tính (linear). Bài toán chúng ta đang làm là một bài toán thuộc loại regression. Bài toán đi tìm các hệ số tối ưu
w_1, w_2, w_3, w_0 chính vì vậy được gọi là bài toán Linear Regression.

Chú ý 1: y là giá trị thực của *outcome* (dựa trên số liệu thống kê chúng ta có trong tập *training data*), trong khi $\hat{y}$ là giá trị mà mô hình Linear Regression dự đoán được. 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.

Chú ý 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 goi là *tuyến tính* nếu đồ thị của nó có dạng một *mặt phẳng*. Trong không gian nhiều hơn 3 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. Chúng ta sẽ được thấy trong các bài viết sau, *tuyến tính* rất quan trọng và hữu ích trong các bài toán Machine Learning. Kinh nghiệm cá nhân tôi cho thấy, trước khi hiểu được các thuật toán *phi tuyến* (non-linear, không phẳng), chúng ta cần nắm vững các kỹ thuật cho các mô hình *tuyến tính*.

3.2 Phân tích toán học

3.2.1 Dạng của Linear Regression

Trong phương trình (1) phía trên, nếu chúng ta đặt $\mathbf{w} = [w_0, w_1, w_2, w_3]^T =$ là vector (cột) hệ số cần phải tối ưu và $\bar{\mathbf{x}} = [1, x_1, x_2, x_3]$ (đọc là x *bar* trong tiếng Anh) là vector (hàng) dữ liệu đầu vào *mở rộng*. Số 1 ở đầu được thêm vào để phép tính đơn giản hơn và thuận tiện cho việc tính toán. Khi đó, phương trình (1) có thể được viết lại dưới dạng:

$$y \approx \bar{\mathbf{x}}\mathbf{w} = \hat{y}$$

Chú ý rằng $\bar{\mathbf{x}}$ là một vector hàng. (Xem thêm về ký hiệu vector hàng và cột tại đây)

3.2.2 Sai số dự đoán

Chúng ta mong muốn rằng sự sai khác e giữa giá trị thực y và giá trị dự đoán $\hat{y}$ (đọc là y *hat* trong tiếng Anh) là nhỏ nhất. Nói cách khác, chúng ta muốn giá trị sau đây càng nhỏ càng tốt:

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

trong đó hệ số $\frac{1}{2}$ (*lại*) là để thuận tiện cho việc tính toán (khi tính đạo hàm thì số $\frac{1}{2}$ sẽ bị triệt tiêu). Chúng ta cần e^2 vì $e = y - \hat{y}$ có thể là một số âm, việc nói e nhỏ nhất sẽ không đúng vì khi $e = -\infty$ là rất nhỏ nhưng sự sai lệch là rất lớn. Bạn đọc có thể tự đặt câu hỏi: **tại sao không dùng trị tuyệt đối $\|e\|$ mà lại dùng bình phương e^2 ở đây?** Câu trả lời sẽ có ở phần sau.

3.2.3 Hàm mất mát

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

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \bar{\mathbf{x}}_\mathbf{i}\mathbf{w})^2 \tag{3.2}$$

Hàm số $\mathcal{L}(\mathbf{w})$ được gọi là **hàm mất mát** (loss function) của bài toán Linear Regression. Chúng ta luôn mong muốn rằng sự mất mát (sai số) là nhỏ nhất, điều đó đồng nghĩa với việc tìm vector hệ số $\mathbf{w}$ sao cho giá trị của hàm mất mát này càng nhỏ càng tốt. Giá trị của $\mathbf{w}$ làm cho hàm mất mát đạt giá trị nhỏ nhất được gọi là *điểm tối ưu* (optimal point), ký hiệu:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w})$$

Trước khi đi tìm lời giải, chúng ta đơn giản hóa phép toán trong phương trình hàm mất mát (3.2). Đặt $\mathbf{y} = [y_1; y_2; \ldots; y_N]$ là một vector cột chứa tất cả các *output* của *training data*; $\bar{\mathbf{X}} = [\bar{\mathbf{x}}_1; \bar{\mathbf{x}}_2; \ldots; \bar{\mathbf{x}}_N]$ là ma trận dữ liệu đầu vào (mở rộng) mà mỗi hàng của nó là một điểm dữ liệu. Khi đó hàm số mất mát $\mathcal{L}(\mathbf{w})$ được viết dưới dạng ma trận đơn giản hơn:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y_i - \bar{\mathbf{x}}_i\mathbf{w})^2 = \frac{1}{2}\|\mathbf{y} - \bar{\mathbf{X}}\mathbf{w}\|_2^2 \tag{3.3}$$

với $\|\mathbf{z}\|_2$ là Euclidean norm (chuẩn Euclid, hay khoảng cách Euclid), nói cách khác $\|\mathbf{z}\|_2^2$ là tổng của bình phương mỗi phần tử của vector $\mathbf{z}$. Tới đây, ta đã có một dạng đơn giản của hàm mất mát được viết như phương trình (3.3).

3.2.4 Nghiệm cho bài toán Linear Regression

Cách phổ biến nhất để tìm nghiệm cho một bài toán tối ưu (chúng ta đã biết từ khi học cấp 3) là giải phương trình đạo hàm (gradient) bằng 0! Tất nhiên đó là khi việc tính đạo hàm và việc giải phương trình đạo hàm bằng 0 không quá phức tạp. Thật may mắn, với các mô hình tuyến tính, hai việc này là khả thi.

Đạo hàm theo $\mathbf{w}$ của hàm mất mát là:

$$\frac{\partial\mathcal{L}(\mathbf{w})}{\partial\mathbf{w}} = \bar{\mathbf{X}}^T(\bar{\mathbf{X}}\mathbf{w} - \mathbf{y})$$

Các bạn có thể tham khảo bảng đạo hàm theo vector hoặc ma trận của một hàm số trong mục D.2 của tài liệu này. *Đến đây tôi xin quay lại câu hỏi ở phần Sai số dự đoán phía trên về việc tại sao không dùng trị tuyệt đối mà lại dùng bình phương. Câu trả lời là hàm bình phương có đạo hàm tại mọi nơi, trong khi hàm trị tuyệt đối thì không (đạo hàm không xác định tại 0).*

Phương trình đạo hàm bằng 0 tương đương với:

$$\bar{\mathbf{X}}^T\bar{\mathbf{X}}\mathbf{w} = \bar{\mathbf{X}}^T\mathbf{y} \triangleq \mathbf{b} \tag{3.4}$$

(ký hiệu $\bar{\mathbf{X}}^T\mathbf{y} \triangleq \mathbf{b}$ nghĩa là *đặt* $\bar{\mathbf{X}}^T\mathbf{y}$ *bằng* $\mathbf{b}$).

Nếu ma trận vuông $\mathbf{A} \triangleq \bar{\mathbf{X}}^T\bar{\mathbf{X}}$ khả nghịch (non-singular hay inversable) thì phương trình (4) có nghiệm duy nhất: $\mathbf{w} = \mathbf{A}^{-1}\mathbf{b}$.

Vậy nếu ma trận $\mathbf{A}$ không khả nghịch (có định thức bằng 0) thì sao? Nếu các bạn vẫn nhớ các kiến thức về hệ phương trình tuyến tính, trong trường hợp này thì hoặc phương trinh (3.4) vô nghiệm, hoặc là nó có vô số nghiệm. Khi đó, chúng ta sử dụng khái niệm *giả nghịch đảo* $\mathbf{A}^\dagger$ (đọc là *A dagger* trong tiếng Anh). (*Giả nghịch đảo (pseudo inverse) là trường hợp tổng quát của nghịch đảo khi ma trận không khả nghịch hoặc thậm chí không vuông. Trong khuôn khổ bài viết này, tôi xin phép được lược bỏ phần này, nếu các bạn thực sự quan tâm, tôi sẽ viết một bài khác chỉ nói về giả nghịch đảo. Xem thêm: Least Squares, Pseudo-Inverses, PCA & SVD.*)

Với khái niệm giả nghịch đảo, điểm tối ưu của bài toán Linear Regression có dạng:

$$\mathbf{w} = \mathbf{A}^\dagger\mathbf{b} = (\bar{\mathbf{X}}^T\bar{\mathbf{X}})^\dagger\bar{\mathbf{X}}^T\mathbf{y} \tag{3.5}$$

3.3 Ví dụ trên Python

3.3.1 Bài toán

Trong phần này, tôi sẽ chọn một ví dụ đơn giản về việc giải bài toán Linear Regression trong Python. Tôi cũng sẽ so sánh nghiệm của bài toán khi giải theo phương trình (3.5) và nghiệm tìm được khi dùng thư viện scikit-learn của Python. (*Đây là thư viện Machine Learning được sử dụng rộng rãi trong Python*). Trong ví dụ này, dữ liệu đầu vào chỉ có 1 giá trị (1 chiều) để thuận tiện cho việc minh hoạ trong mặt phẳng.

Chúng ta có 1 bảng dữ liệu về chiều cao và cân nặng của 15 người như trong Bảng 3.1.

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*). Vì blog này nói về các thuật toán Machine Learning đơn giản nên tôi sẽ giả sử rằng chúng ta có thể dự đoán được.

Table 3.1: 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		

Chúng ta có thể thấy là cân nặng sẽ tỉ lệ thuận với chiều cao (càng cao càng nặng), nên có thể sử dụng Linear Regression model cho việc dự đoán này. Để kiểm tra độ chính xác của model tìm được, chúng ta sẽ giữ lại cột 155 và 160 cm để kiểm thử, các cột còn lại được sử dụng để huấn luyện (train) model.

3.3.2 Hiển thị dữ liệu trên đồ thị

Trước tiên, chúng ta cần có hai thư viện numpy cho đại số tuyến tính và matplotlib cho việc vẽ hình.

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib.pyplot as plt
```

Tiếp theo, chúng ta khai báo và biểu diễn dữ liệu trên một đồ thị.

```
# height (cm)
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]]).T
# Visualize data
plt.plot(X, y, 'ro')
plt.axis([140, 190, 45, 75])
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
plt.show()
```

Từ đồ thị trong Hình 3.1 ta thấy rằng dữ liệu được sắp xếp gần như theo 1 đường thẳng, vậy mô hình Linear Regression nhiều khả năng sẽ cho kết quả tốt:

(cân nặng) = `w_1`*(chiều cao) + `w_0`

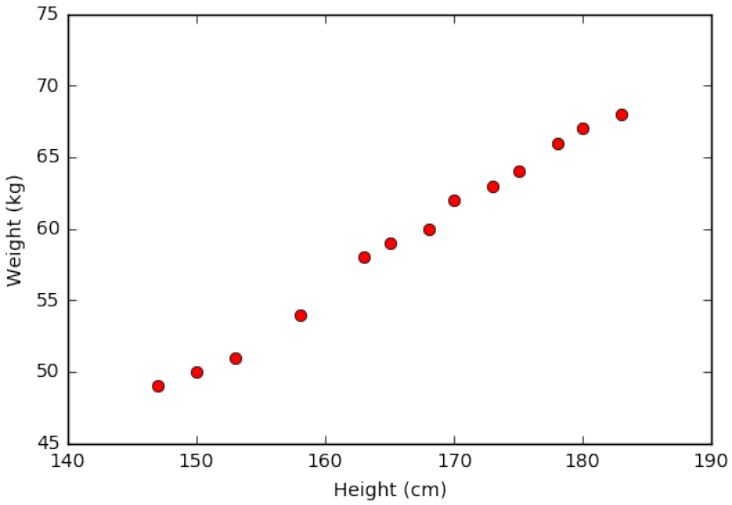

Fig. 3.1: Phân bố của dữ liệu trong ví dụ có dạng đường thẳng

3.3.3 Nghiệm theo công thức

Tiếp theo, chúng ta sẽ tính toán các hệ số `w_1` và `w_0` dựa vào công thức (5). Chú ý: giả nghịch đảo của một ma trận `A` trong Python sẽ được tính bằng `numpy.linalg.pinv(A)`, `pinv` là từ viết tắt của *pseudo inverse*.

```
# Building Xbar
one = np.ones((X.shape[0], 1))
Xbar = np.concatenate((one, X), axis = 1)

# 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)
print('w = ', w)
# Preparing the fitting line
w_0 = w[0][0]
w_1 = w[1][0]
x0 = np.linspace(145, 185, 2)
y0 = w_0 + w_1*x0

# Drawing the fitting line
plt.plot(X.T, y.T, 'ro')     # data
plt.plot(x0, y0)               # the fitting line
plt.axis([140, 190, 45, 75])
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
plt.show()
```

```
w =  [[-33.73541021]
 [  0.55920496]]
```

Từ đồ thị bên trên 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 *training*. Bây giờ,

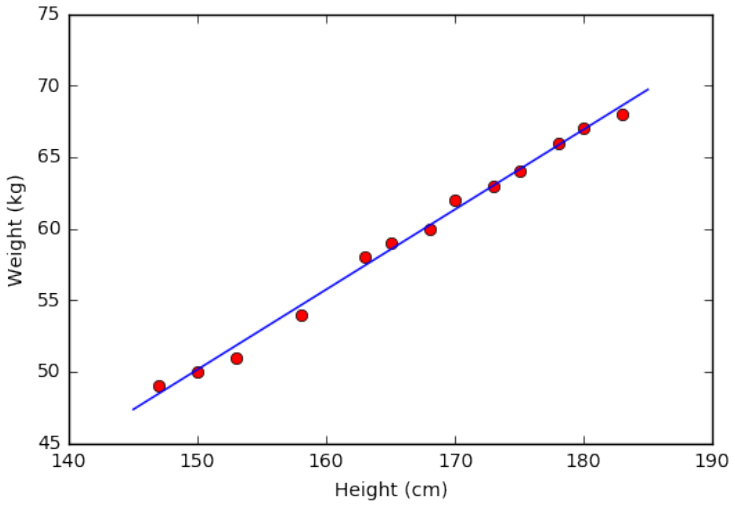

Fig. 3.2: Minh họa nghiệm của ví dụ

chúng ta sử dụng mô hình này để dự đoán cân nặng của hai người có chiều cao 155 và 160 cm mà chúng ta đã không dùng khi tính toán nghiệm.

```
y1 = w_1*155 + w_0
y2 = w_1*160 + w_0

print( 'Predict weight of person 155 cm: %.2f (kg), real number: 52 (kg)'  %(y1) )
print( 'Predict weight of person 160 cm: %.2f (kg), real number: 56 (kg)'  %(y2) )
```

```
Predict weight of person 155 cm: 52.94 (kg), real number: 52 (kg)
Predict weight of person 160 cm: 55.74 (kg), real number: 56 (kg)
```

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

3.3.4 Nghiệm theo thư viện scikit-learn

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

```
from sklearn import datasets, linear_model

# fit the model by Linear Regression
regr = linear_model.LinearRegression(fit_intercept=False) # fit_intercept = False for
    calculating the bias
regr.fit(Xbar, y)

# Compare two results
print( 'Solution found by scikit-learn  : ', regr.coef_ )
print( 'Solution found by (5): ', w.T)
```

```
    Solution found by scikit-learn  :  [[  -33.73541021 0.55920496]]
    Solution found by (5):  [[  -33.73541021 0.55920496 ]]
```

Chúng ta thấy rằng hai kết quả thu được như nhau! (*Nghĩa là tôi đã không mắc lỗi nào trong cách tìm nghiệm ở phần trên*)

Source code Jupyter Notebook cho bài này.

3.4 Thảo luận

3.4.1 Các bài toán có thể giải bằng Linear Regression

Hàm số $y \approx f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ 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:

$$y \approx w_1x_1 + w_2x_2 + w_3x_1^2 + w_4\sin(x_2) + w_5x_1x_2 + w_0$$

là một hàm tuyến tính theo $\mathbf{w}$ và vì vậy cũng có thể được giải bằng Linear Regression. Với mỗi dữ liệu đầu vào $\mathbf{x} = [x_1; x_2]$, chúng ta tính toán dữ liệu mới $\tilde{\mathbf{x}} = [x_1, x_2, x_1^2, \sin(x_2), x_1x_2]$ (đọc là x *tilde* trong tiếng Anh) rồi áp dụng Linear Regression với dữ liệu mới này.

Xem thêm ví dụ về Quadratic Regression (Hồi Quy Bậc Hai).

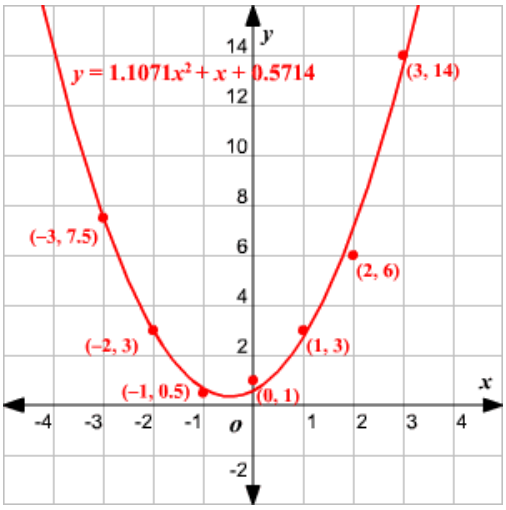

Fig. 3.3: Quadratic Regression (Nguồn: Quadratic Regression)

3.4.2 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 *nhiễu* (150 cm, 90kg) thì kết quả sẽ sai khác đi rất nhiều (Xem Hình 3.4).

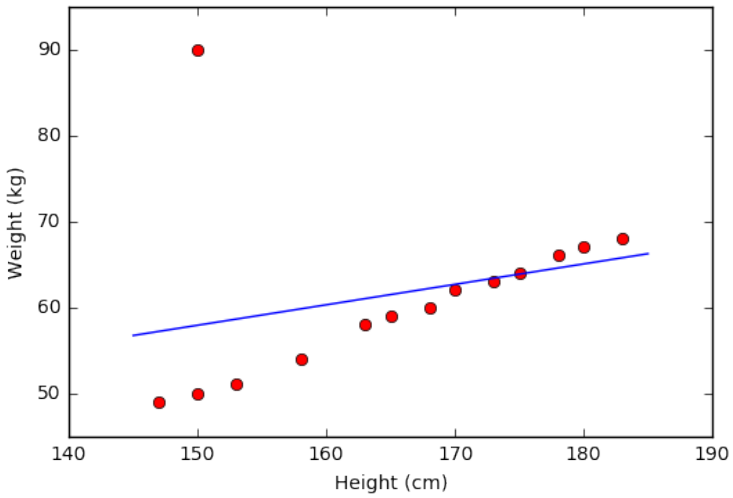

Fig. 3.4: Linear Regression rất nhạy cảm với nhiễu

Vì vậy, trước khi thực hiện Linear Regression, các nhiễu (*outlier*) cần phải được loại bỏ. Bước này được gọi là tiền xử lý (pre-processing).

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, chúng ta sẽ tự hỏi: làm thế nào để xác định được các hàm $x_1^2, \sin(x_2), x_1 x_2$ như ở trên?!

3.4.3 Các phương pháp tối ưu

Linear Regression là một mô hình đơn giản, lời giải cho phương trình đạo hàm bằng 0 cũng khá đơn giản. *Trong hầu hết các trường hợp, chúng ta không thể giải được phương trình đạo hàm bằng 0.*

Nhưng có một điều chúng ta nên nhớ, **còn tính được đạo hàm là còn có cơ hội**.

3.5 Tài liệu tham khảo

1. Linear Regression - Wikipedia
2. Simple Linear Regression Tutorial for Machine Learning
3. Least Squares, Pseudo-Inverses, PCA & SVD

4

K-means Clustering

4.1 Giới thiệu

Trong bài trước, chúng ta đã làm quen với thuật toán Linear Regression - là thuật toán đơn giản nhất trong Supervised learning. Bài này tôi sẽ giới thiệu một trong những thuật toán cơ bản nhất trong Unsupervised learning - thuật toán K-means clustering (phân cụm K-means).

Trong thuật toán K-means clustering, chúng ta không biết nhãn (label) của từng điểm dữ liệu. Mục đích là làm thể nào để phân dữ liệu thành các cụm (cluster) khác nhau sao cho *dữ liệu trong cùng một cụm có tính chất giống nhau.*

Ví dụ: Một công ty muốn tạo ra những 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; ...). Giả sử công ty đó có rất nhiều dữ liệu của rất nhiều khách hàng nhưng chưa có cách nào chia toàn bộ khách hàng đó thành một số nhóm/cụm khác nhau. Nếu một người biết Machine Learning được đặt câu hỏi này, phương pháp đầu tiên anh (chị) ta nghĩ đến sẽ là K-means Clustering. Vì nó là một trong những thuật toán đầu tiên mà anh ấy tìm được trong các cuốn sách, khóa học về Machine Learning. Và tôi cũng chắc rằng anh ấy đã đọc blog Machine Learning cơ bản. 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.

Ý tưởng đơn giản nhất về cluster (cụm) là tập hợp các điểm *ở gần nhau trong một không gian nào đó* (không gian này có thể có rất nhiều chiều trong trường hợp thông tin về một điểm dữ liệu là rất lớn). Hình bên dưới là một ví dụ về 3 cụm dữ liệu (từ giờ rôi sẽ viết gọn là *cluster*).

Giả sử mỗi cluster có một điểm đại diện (*center*) màu vàng. Và những điểm xung quanh mỗi center thuộc vào cùng nhóm với center đó. Một cách đơn giản nhất, xét một điểm bất kỳ, ta xét xem điểm đó gần với center nào nhất thì nó thuộc về cùng nhóm với center đó.

Tới đây, chúng ta có một bài toán thú vị: *Trên một vùng biển hình vuông lớn có ba đảo hình vuông, tam giác, và tròn màu vàng như hình trên. 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 của các đảo.*

Hình dưới đây là một hình minh họa cho việc phân chia lãnh hải nếu có 5 đảo khác nhau được biểu diễn bằng các hình tròn màu đen:

Chúng ta thấy rằng đường phân định giữa các lãnh hải là các đường thẳng (chính xác hơn thì chúng là các đường trung trực của các cặp điểm gần nhau). Vì vậy, lãnh hải của một đảo sẽ là một hình đa giác.

Cách phân chia này trong toán học được gọi là Voronoi Diagram.

Trong không gian ba chiều, lấy ví dụ là các hành tinh, thì (tạm gọi là) *lãnh không* của mỗi hành tinh sẽ là một đa diện. Trong không gian nhiều chiều hơn, chúng ta sẽ có những thứ (mà tôi gọi là) *siêu đa diện* (hyperpolygon).

Quay lại với bài toán phân nhóm và cụ thể là thuật toán K-means clustering, chúng ta cần một chút phân tích toán học trước khi đi tới phần tóm tắt thuật toán ở phần dưới. Nếu bạn không muốn đọc quá nhiều về toán, bạn có thể bỏ qua phần này. (*Tốt nhất là đừng bỏ qua, bạn sẽ tiếc đấy*).

4.2 Phân tích toán học

Mục đích cuối cùng của thuật toán phân nhóm này là: từ dữ liệu đầu vào và số lượng nhóm chúng ta muốn tìm, hãy chỉ ra center 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.

4.2.1 Một số ký hiệu toán học

Giả sử có N điểm dữ liệu là $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ và $K < N$ là số cluster chúng ta muốn phân chia. Chúng ta cần tìm các center $\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_K \in \mathbb{R}^{d \times 1}$ và label của mỗi điểm dữ liệu.

Lưu ý về ký hiệu toán học: *trong các bài viết của tôi, các số vô hướng được biểu diễn bởi các chữ cái viết ở dạng không in đậm, có thể viết hoa, ví dụ x_1, N, y, k. Các vector được biểu diễn bằng các chữ cái thường in đậm, ví dụ $\mathbf{m}, \mathbf{x}_1$. Các ma trận được biểu diễn bởi các chữ viết hoa in đậm, ví dụ $\mathbf{X}, \mathbf{M}, \mathbf{Y}$. Lưu ý này đã được nêu ở bài Linear Regression. Tôi xin được không nhắc lại trong các bài tiếp theo.*

Với mỗi điểm dữ liệu $\mathbf{x}_i$ đặt $\mathbf{y}_i = [y_{i1}, y_{i1}, \ldots, y_{iK}]$ là label vector của nó, trong đó nếu $\mathbf{x}_i$ được phân vào cluster k thì $y_{ik} = 1$ và $y_{ij} = 0, \forall j \neq k$. Điều này có nghĩa là có đúng một phần tử của vector $\mathbf{y}_i$ là bằng 1 (tương ứng với cluster của $\mathbf{x}_i$), các phần tử còn lại bằng 0. Ví dụ: nếu một điểm dữ liệu có label vector là $[1, 0, 0, \ldots, 0]$ thì nó thuộc vào cluster 1, là $[0, 1, 0, \ldots, 0]$ thì nó thuộc vào cluster 2, ... Cách mã hóa label của dữ liệu như thế này được goi là biểu diễn *one-hot*. Chúng ta sẽ thấy cách biểu diễn one-hot này rất phổ biến trong Machine Learning ở các bài tiếp theo.

Ràng buộc của $\mathbf{y}_i$ có thể viết dưới dạng toán học như sau:

$$y_{ik} \in 0, 1, \quad \sum_{k=1}^{K} y_{ik} = 1 \quad (1) \tag{4.1}$$

4.2.2 Hàm mất mát và bài toán tối ưu

Nếu ta coi center $\mathbf{m}_k$ là center (hoặc representative) của mỗi cluster và *ước lượng* tất cả các điểm được phân vào cluster này bởi $\mathbf{m}_k$, thì 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 sai số này có trị tuyệt đối nhỏ nhất nên (giống như trong bài Linear Regression) ta sẽ tìm cách để đại lượng sau đây đạt giá trị nhỏ nhất:

$$\|\mathbf{x}_i - \mathbf{m}_k\|_2^2 \tag{4.2}$$

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 bên trên sẽ được viết lại là:

$$y_{ik}\|\mathbf{x}_i - \mathbf{m}_k\|_2^2 = \sum_{j=1}^{K} y_{ij}\|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \tag{4.3}$$

(*Hy vọng chỗ này không quá khó hiểu*)

Sai số cho toàn bộ dữ liệu sẽ là:

$$\mathcal{L}(\mathbf{Y}, \mathbf{M}) = \sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \tag{4.4}$$

Trong đó $\mathbf{Y} = [\mathbf{y}_1; \mathbf{y}_2; \ldots; \mathbf{y}_N]$, $\mathbf{M} = [\mathbf{m}_1, \mathbf{m}_2, \ldots \mathbf{m}_K]$ lần lượt là các ma trận được tạo bởi label vector của mỗi điểm dữ liệu và center của mỗi cluster. Hàm số mất mát trong bài toán K-means clustering của chúng ta là hàm $\mathcal{L}(\mathbf{Y}, \mathbf{M})$ với ràng buộc như được nêu trong phương trình (1).

Tóm lại, chúng ta cần tối ưu bài toán sau:

$$\mathbf{Y}, \mathbf{M} = \arg\min_{\mathbf{Y},\mathbf{M}} \sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (2) \tag{4.5}$$

$$\text{subject to: } y_{ij} \in 0, 1 \ \ \forall i, j; \quad \sum_{j=1}^{K} y_{ij} = 1 \ \ \forall i \tag{4.6}$$

(*subject to* nghĩa là *thỏa mãn điều kiện*).

Nhắc lại khái niệm arg min: Chúng ta biết ký hiệu min là *giá trị nhỏ nhất của hàm số*, arg min chính là *giá trị của biến số để hàm số đó đạt giá trị nhỏ nhất đó*. Nếu $f(x) = x^2 - 2x + 1 = (x-1)^2$ thì giá trị nhỏ nhất của hàm số này bằng 0, đạt được khi $x = 1$. Trong ví dụ này $\min_x f(x) = 0$ và $\arg\min_x f(x) = 1$. Thêm ví dụ khác, nếu $x_1 = 0, x_2 = 10, x_3 = 5$ thì ta nói $\arg\min_i x_i = 1$ vì 1 là chỉ số để x_i đạt giá trị nhỏ nhất (bằng 0). Biến số viết bên dưới min là biến số cúng ta cần tối ưu. Trong các bài toán tối ưu, ta thường quan tâm tới arg min hơn là min.

4.2.3 Thuật toán tối ưu hàm mất mát

Bài toán (2) 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 (global optimal point, tức nghiệm làm cho hàm mất mát đạt giá trị nhỏ nhất có thể).* Tuy nhiên, trong một số trường hợp chúng ta vẫn có thể tìm được phương pháp để tìm được nghiệm gần đúng hoặc điểm cực tiểu. (*Nếu chúng ta vẫn nhớ chương trình toán ôn thi đại học thì điểm cực tiểu chưa chắc đã phải là điểm làm cho hàm số đạt giá trị nhỏ nhất*).

Một cách đơn giản để giải bài toán (2) là xen kẽ giải $\mathbf{Y}$ và $\mathbf{M}$ khi biến còn lại được cố định. Đây là một thuật toán lặp, cũng là kỹ thuật phổ biến khi giải bài toán tối ưu. Chúng ta sẽ lật lượt giải quyết hai bài toán sau đây:

Cố định M, tìm Y

Giả sử đã tìm được các centers, 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 centers 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:

$$\mathbf{y}_i = \arg\min_{\mathbf{y}_i} \sum_{j=1}^{K} y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \quad (3) \tag{4.7}$$

$$\text{subject to: } y_{ij} \in 0, 1 \ \ \forall j; \quad \sum_{j=1}^{K} y_{ij} = 1 \tag{4.8}$$

Vì chỉ có một phần tử của label vector $\mathbf{y}_i$ bằng 1 nên bài toán (3) có thể tiếp tục được viết dưới dạng đơn giản hơn:

$$j = \arg\min_j \|\mathbf{x}_i - \mathbf{m}_j\|_2^2 \tag{4.9}$$

Vì $\|\mathbf{x}_i - \mathbf{m}_j\|_2^2$ chính là bình phương khoảng cách tính từ điểm $\mathbf{x}_i$ tới center $\mathbf{m}_j$, ta có thể kết luận rằng **mỗi điểm $\mathbf{x}_i$ thuộc vào cluster có center gần nó nhất**! Từ đó ta có thể dễ dàng suy ra label vector của từng điểm dữ liệu.

Cố định Y, tìm M

Giả sử đã tìm được cluster cho từng điểm, hãy tìm center mới cho mỗi cluster để hàm mất mát đạt giá trị nhỏ nhất.

Một khi chúng ta đã xác định được label vector cho từng điểm dữ liệu, bài toán tìm center cho mỗi cluster được rút gọn thành:

$$\mathbf{m}_j = \arg\min_{\mathbf{m}_j} \sum_{i=1}^{N} y_{ij} \|\mathbf{x}_i - \mathbf{m}_j\|_2^2. \tag{4.10}$$

Tới đây, ta có thể tìm nghiệm bằng phương pháp giải đạo hàm bằng 0, vì hàm cần tối ưu là một hàm liên tục và có đạo hàm xác định tại mọi điểm. *Và quan trọng hơn, hàm này là hàm convex (lồi) theo $\mathbf{m}_j$ nên chúng ta sẽ tìm được giá trị nhỏ nhất và điểm tối ưu tương ứng. Sau này nếu có dịp, tôi sẽ nói thêm về tối ưu lồi (convex optimization) - một mảng cực kỳ quan trọng trong toán tối ưu.*

Đặt $l(\mathbf{m}_j)$ là hàm bên trong dấu arg min, ta có đạo hàm:

$$\frac{\partial l(\mathbf{m}_j)}{\partial \mathbf{m}_j} = 2 \sum_{i=1}^{N} y_{ij} (\mathbf{m}_j - \mathbf{x}_i) \tag{4.11}$$

Giải phương trình đạo hàm bằng 0 ta có:

$$\mathbf{m}_j \sum_{i=1}^{N} y_{ij} = \sum_{i=1}^{N} y_{ij} \mathbf{x}_i \tag{4.12}$$

$$\Rightarrow \mathbf{m}_j = \frac{\sum_{i=1}^{N} y_{ij} \mathbf{x}_i}{\sum_{i=1}^{N} y_{ij}} \tag{4.13}$$

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 (*Bạn có nhận ra không?*). Còn tử số chính là *tổng các điểm dữ liệu* trong

cluster j. (*Nếu bạn đọc vẫn nhớ điều kiện ràng buộc của các y_{ij} thì sẽ có thể nhanh chóng nhìn ra điều này*).

Hay nói một cách đơn giản hơn nhiều: **$\mathbf{m}_j$ là trung bình cộng của các điểm trong cluster j.**

Tên gọi *K-means clustering* cũng xuất phát từ đây.

4.2.4 Tóm tắt thuật toán

Tới đây tôi xin được tóm tắt lại thuật toán (*đặc biệt quan trọng với các bạn bỏ qua phần toán học bên trên*) như sau:

Đầu vào: Dữ liệu $\mathbf{X}$ và số lượng cluster cần tìm K.

Đầu ra: Các center $\mathbf{M}$ và label vector cho từng điểm dữ liệu $\mathbf{Y}$.

1. Chọn K điểm bất kỳ làm các center ban đầu. 2. Phân mỗi điểm dữ liệu vào cluster có center gần nó nhất. 3. Nếu việc gán 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 center cho từng cluster bằng cách lấy trung bình cộng của tất các các điểm dữ liệu đã được gán vào cluster đó sau bước 2. 5. Quay lại bước 2.

Chúng ta có thể đảm bảo rằng thuật toán sẽ dừng lại 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. Theo kiến thức về dãy số trong chương trình cấp 3: *nếu một dãy số giảm và bị chặn dưới thì nó hội tụ!* Hơn nữa, số lượng cách phân nhóm cho toàn bộ dữ liệu là hữu hạn nên đến một lúc nào đó, hàm mất mát sẽ không thể thay đổi, và chúng ta có thể dừng thuật toán tại đây.

Chúng ta sẽ có một vài thảo luận về thuật toán này, về những hạn chế và một số phương pháp khắc phục. Nhưng trước hết, hãy xem nó thể hiện như thế nào trong một ví dụ cụ thể dưới đây.

4.3 Ví dụ trên Python

4.3.1 Giới thiệu bài toán

Để kiểm tra mức độ hiểu quả của một thuật toán, chúng ta sẽ làm một ví dụ đơn giản (thường được gọi là *toy example*). Trước hết, chúng ta chọn center cho từng cluster và tạo dữ liệu cho từng cluster bằng cách lấy mẫu theo phân phối chuẩn có kỳ vọng là center của cluster đó và ma trận hiệp phương sai (covariance matrix) là ma trận đơn vị.

Trước tiên, chúng ta cần khai báo các thư viện cần dùng. Chúng ta cần `numpy` và `matplotlib` như trong bài Linear Regression cho việc tính toán ma trận và hiển thị dữ liệu. Chúng ta cũng cần thêm thư viện `scipy.spatial.distance` để tính khoảng cách giữa các cặp điểm trong hai tập hợp một cách hiệu quả.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
np.random.seed(11)
```

Tiếp theo, ta tạo dữ liệu bằng cách lấy các điểm theo phân phối chuẩn có kỳ vọng tại các điểm có tọa độ `(2, 2)`, `(8, 3)` và `(3, 6)`, ma trận hiệp phương sai giống nhau và là ma trận đơn vị. Mỗi cluster có 500 điểm. (*Chú ý rằng mỗi điểm dữ liệu là một hàng của ma trận dữ liệu.*

```
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

original_label = np.asarray([0]*N + [1]*N + [2]*N).T
```

4.3.2 Hiển thị dữ liệu trên đồ thị

Chúng ta cần một hàm `kmeans_display` để hiển thị dữ liệu. Sau đó hiển thị dữ liệu theo nhãn ban đầu.

```
def kmeans_display(X, label):
    K = np.amax(label) + 1
    X0 = X[label == 0, :]
    X1 = X[label == 1, :]
    X2 = X[label == 2, :]

    plt.plot(X0[:, 0], X0[:, 1], 'b^', markersize = 4, alpha = .8)
    plt.plot(X1[:, 0], X1[:, 1], 'go', markersize = 4, alpha = .8)
    plt.plot(X2[:, 0], X2[:, 1], 'rs', markersize = 4, alpha = .8)

    plt.axis('equal')
    plt.plot()
    plt.show()

kmeans_display(X, original_label)
```

Trong đồ thị trên, mỗi cluster tương ứng với một màu. Có thể nhận thấy rằng có một vài điểm màu đỏ bị lẫn sang phần cluster màu xanh.

4.3.3 Các hàm số cần thiết cho K-means clustering

Viết các hàm:

1. `kmeans_init_centers` để khởi tạo các centers ban đầu. 2. `kmeans_asign_labels` để gán nhán mới cho các điểm khi biết các centers. 3. `kmeans_update_centers` để cập nhật các centers mới dữa trên dữ liệu vừa được gán nhãn. 4. `has_converged` để kiểm tra điều kiện dừng của thuật toán.

```
def kmeans_init_centers(X, k):
    # randomly pick k rows of X as initial centers
    return X[np.random.choice(X.shape[0], k, replace=False)]

def kmeans_assign_labels(X, centers):
    # calculate pairwise distances btw data and centers
    D = cdist(X, centers)
    # return index of the closest center
    return np.argmin(D, axis = 1)

def kmeans_update_centers(X, labels, K):
    centers = np.zeros((K, X.shape[1]))
    for k in range(K):
        # collect all points assigned to the k-th cluster
        Xk = X[labels == k, :]
        # take average
        centers[k,:] = np.mean(Xk, axis = 0)
    return centers

def has_converged(centers, new_centers):
    # return True if two sets of centers are the same
    return (set([tuple(a) for a in centers]) ==
        set([tuple(a) for a in new_centers]))
```

Phần chính của K-means clustering:

```
def kmeans(X, K):
    centers = [kmeans_init_centers(X, K)]
    labels = []
    it = 0
    while True:
        labels.append(kmeans_assign_labels(X, centers[-1]))
        new_centers = kmeans_update_centers(X, labels[-1], K)
        if has_converged(centers[-1], new_centers):
            break
        centers.append(new_centers)
        it += 1
    return (centers, labels, it)
```

Áp dụng thuật toán vừa viết vào dữ liệu ban đầu, hiển thị kết quả cuối cùng.

```
(centers, labels, it) = kmeans(X, K)
print 'Centers found by our algorithm:'
print centers[-1]

kmeans_display(X, labels[-1])
```

Centers found by our algorithm: [[1.97563391 2.01568065] [8.03643517 3.02468432] [2.99084705 6.04196062]]

Từ kết quả này chúng ta thấy rằng thuật toán K-means clustering làm việc khá thành công, các centers tìm được khá gần với kỳ vọng ban đầu. Các điểm thuộc cùng một cluster hầu như được phân vào cùng một cluster (trừ một số diểm màu đỏ ban đầu đã bị phân nhầm vào cluster màu xanh da trời, nhưng tỉ lệ là nhỏ và có thể chấp nhận được).

Dưới đây là hình ảnh động minh họa thuật toán qua từng vòng lặp, chúng ta thấy rằng thuật toán trên hội tụ rất nhanh, chỉ cần 6 vòng lặp để có được kết quả cuối cùng: <img src ="/assets/kmeans/kmeans11.gif" align = "center">

Các bạn có thể xem thêm các trang web minh họa thuật toán K-means cluster tại:

1. Visualizing K-Means Clustering
2. Visualizing K-Means Clustering - Standford

4.3.4 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
kmeans = KMeans(n_clusters=3, random_state=0).fit(X)
print 'Centers found by scikit-learn:'
print kmeans.cluster_centers_
pred_label = kmeans.predict(X)
kmeans_display(X, pred_label)
```

Centers found by scikit-learn: [[8.0410628 3.02094748] [2.99357611 6.03605255] [1.97634981 2.01123694]]

Thật may mắn (*cho tôi*), hai thuật toán cho cùng một đáp số! Với cách thứ nhất, tôi mong muốn các bạn hiểu rõ được thuật toán K-means clustering làm việc như thế nào. Với cách thứ hai, tôi hy vọng các bạn biết áp dụng thư viện sẵn có như thế nào.

4.4 Thảo luận

4.4.1 Hạn chế

Có một vài hạn chế của thuật toán K-means clustering:

Chúng ta cần biết số lượng cluster cần clustering

Để ý thấy rằng trong thuật toán nêu trên, chúng ta cần biết đại lượng K là số lượng clusters. Trong thực tế, nhiều trường hợp chúng ta không xác định được giá trị này. Có một số phương pháp giúp xác định số lượng clusters, tôi sẽ dành thời gian nói về chúng sau nếu có dịp. Bạn đọc có thể tham khảo Elbow method - Determining the number of clusters in a data set.

Nghiệm cuối cùng phụ thuộc vào các centers được khởi tạo ban đầu

Tùy vào các center ban đầu mà thuật toán có thể có tốc độ hội tụ rất chậm, ví dụ: hoặc thậm chí cho chúng ta nghiệm không chính xác (chỉ là local minimum - điểm cực tiểu - mà không phải giá trị nhỏ nhất):

Có một vài cách khắc phục đó là:

- Chạy K-means clustering nhiều lần với các center ban đầu khác nhau rồi chọn cách có hàm mất mát cuối cùng đạt giá trị nhỏ nhất.
- K-means++ -Improve initialization algorithm - wiki.
- Bạn nào muốn tìm hiểu sâu hơn có thể xem bài báo khoa học Cluster center initialization algorithm for K-means clustering.

Các cluster cần có só lượng điểm gần bằng nhau

Dưới đây là một ví dụ với 3 cluster với 20, 50, và 1000 điểm. Kết quả cuối cùng không chính xác.

Các cluster cần có dạng hình tròn

Tức các cluster tuân theo phân phối chuẩn và ma trận hiệp phương sai là ma trận đường chéo có các điểm trên đường chéo giống nhau.

Dưới đây là 1 ví dụ khi 1 cluster có dạng hình dẹt.

Khi một cluster nằm phía trong 1 cluster khác

Đây là ví dụ kinh điển về việc K-means clustering không thể phân cụm dữ liệu. Một cách tự nhiên, chúng ta sẽ phân ra thành 4 cụm: mắt trái, mắt phải, miệng, xunh quanh mặt. Nhưng vì mắt và miệng nằm trong khuôn mặt nên K-means clustering không thực hiện được:

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 sau này. Chúng ta cần bắt đầu từ những thứ đơn giản. *Simple is best!*

4.5 Tài liệu tham khảo

1. Clustering documents using k-means
2. Voronoi Diagram - Wikipedia
3. Cluster center initialization algorithm for K-means clustering
4. Visualizing K-Means Clustering
5. Visualizing K-Means Clustering - Standford

5

K-means Clustering: Simple Applications

Trong bài này, tôi sẽ áp dụng thuật toán K-means clustering vào ba bài toán xử lý ảnh thực tế hơn: i) Phân nhóm các chữ số viết tay, ii) Tách vật thể (image segmentation) và iii) Nén ảnh/dữ liệu (image compression). Qua đây, tôi cũng muốn độc giả làm quen với một số kỹ thuật đơn giản trong xử lý hình ảnh - một mảng quan trọng trong Machine Learning. Souce code cho các ví dụ trong trang này có thể được tìm thấy tại đây.

5.1 Phân nhóm chữ số viết tay

5.1.1 Bộ cơ sở dữ liệu MNIST

Bộ cơ sở dữ liệu MNIST là bộ cơ sở dữ liệu lớn nhất về chữ số viết tay và được sử dụng trong hầu hết các thuật toán nhận dạng hình ảnh (Image Classification).

MNIST bao gồm hai tập con: tập dữ liệu huấn luyện (training set) có tổng cộng 60k ví dụ khác nhau về chữ số viết tay từ 0 đến 9, tập dữ liệu kiểm tra (test set) có 10k ví dụ khác nhau. Tất cả đều đã được gán nhãn. Hình dưới đây là ví dụ về một số hình ảnh được trích ra từ MNIST.

Mỗi bức ảnh là một ảnh đen trắng (có 1 channel), có kích thước 28x28 pixel (tổng cộng 784 pixels). Mỗi pixel mang một giá trị là một số tự nhiên từ 0 đến 255. Các pixel màu đen có giá trị bằng 0, các pixel càng trắng thì có giá trị càng cao (nhưng không quá 255). Dưới đây 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, tôi đã resize bức ảnh về 14x14*)

5.1.2 Bài toán phân nhóm giả định

Bài toán: Giả sử rằng chúng ta không biết nhãn của các chữ số này, chúng ta muốn phân nhóm các bức ảnh gần giống nhau về một nhóm.

Lại thêm một giả sử nữa là chúng ta mới chỉ biết tới thuật toán phân nhóm K-means clustering gần đây từ blog Machine Learning cơ bản (*xin lỗi độc giả vì để các bài học có ý nghĩa hơn, chúng ta đôi khi cần những giả định không được thực tế cho lắm*), chúng ta sẽ giải quyết bài toán này thế nào?

Trước khi áp dụng thuật toán K-means clustering, chúng ta cần coi mỗi bức ảnh là một điểm dữ liệu. Và vì mỗi điểm dữ liệu là 1 vector (hàng hoặc cột) chứ không phải ma trận như số 7 ở trên, chúng ta phải làm thêm một bước đơn giản trung gian gọi là *vectorization* (vector hóa). Nghĩa là, để có được 1 vector, ta có thể tách các hàng của ma trận pixel ra, sau đó đặt chúng cạnh nhau, và chúng ta được một vector hàng rất dài biểu diễn 1 bức ảnh chữ số.

Chú ý: *Cách làm này chỉ là cách đơn giản nhất để mô tả dữ liệu ảnh bằng 1 vector. Trên thực tế, người ta áp dụng rất nhiều kỹ thuật khác nhau để có thể tạo ra các vector đặc trưng (feature vector) giúp các thuật toán có được kết quả tốt hơn.*

5.1.3 Làm việc trên Python

Trước tiên các bạn vào trang chủ của MNIST để download bộ cơ sở dữ liệu này. Mặc dù trong bài này chúng ta chỉ dùng bộ dữ liệu test với 10k ảnh và không cần label, các bạn vẫn cần download cả hai file `t10k-images-idx3-ubyte.gz` và `t10k-labels-idx1-ubyte.gz` vì thư viện `python-mnist` cần cả hai file này để load dữ liệu từ tập test.

Trước tiên chúng ta cần khai báo một số thư viện:

`numpy` cho các phép toán liên quan đến ma trận. `mnist` để đọc dữ liệu từ MNIST. `matplotlib` để hiển thị hình vẽ. `sklearn` chính là `scikit-learn` mà chúng ta đã làm quen trong các bài trước. (*Về việc cài đặt các thư viện này, tôi hy vọng bạn đọc có thể Google thêm đôi chút. Nếu có khó khăn trong việc cài đặt, hãy để lại comment ở dưới bài. Lưu ý, làm việc trên Windows sẽ khó khăn hơn một chút so với Linux*)

```
# %reset
import numpy as np
from mnist import MNIST # require 'pip install python-mnist'
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

Để hiện thị nhiều bức ảnh các chữ số cùng một lúc, tôi có dùng thêm hàm số `display_network.py`.

Thực hiện thuật toán K-means clustering trên toàn bộ 10k chữ số.

```
from display_network import *

mndata = MNIST('../MNIST/') # path to your MNIST folder
mndata.load_testing()
X = mndata.test_images

kmeans = KMeans(n_clusters=K).fit(X)
pred_label = kmeans.predict(X)
```

(*Phần còn lại của source code có thể được tìm thấy tại đây*)

Đến đây, sau khi đã tìm được các center và phân nhóm dữ liệu vào từng cluster, tôi muốn hiển thị xem center trông như thế nào và các bức ảnh được phân vào mỗi cluster có giống nhau hay không. Dưới đây là kết quả khi tôi chọn ngẫu nhiên 20 bức ảnh từ mỗi cluster.

Mỗi hàng tương ứng với một cluster, cột đầu tiên có nền xanh bên trái là centers tìm được của các clusters (màu đỏ hơn là các pixel có giá trị cao hơn). Chúng ta thấy rằng các center đều hoặc là giống với một chữ số nào đó, hoặc là kết hợp của hai/ba chữ số nào đó. Ví dụ: center của nhóm thứ 4 là sự kết hợp của các số 4, 7, 9; của hàng thứ 7 là kết hợp của chữ số 7, 8 và 9.

Tuy nhiên, các bức ảnh lấy ra ngẫu nhiên từ mỗi nhóm trông không thực sự giống nhau. Lý do có thể là những bức ảnh này ở xa các center của mỗi nhóm (mặc dù center đó đã là gần nhất). Như vậy thuật toán K-means clustering làm việc không thực sự tốt trong trường hợp này. (*Thật may là vì thế nên chúng ta vẫn còn nhiều thứ để học nữa*).

Chúng ta vẫn có thể khai thác một số thông tin hữu ích sau khi thực hiện thuật toán này. Bây giờ, thay vì chọn ngẫu nhiên các bức ảnh trong mỗi cluster, tôi chọn 20 bức ảnh gần center của mỗi cluster nhất, vì càng gần center thì độ tin cậy càng cao. Hãy xem hình dưới đây:

Bạn đọc có thể thấy dữ liệu trong mỗi hàng khá giống nhau và giống với center 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, một thẳng, một chéo. Và K-means clustering nghĩ rằng đó là hai chữ số khác nhau. Điều này là dễ hiểu vì K-means clustering là thuật toán Unsupervised learning. Nếu có sự can thiệp của con người, chúng ta có thể nhóm hai clusters này vào làm một.

2. Hàng số 9, chữ số 4 và 9 được phân vào cùng 1 cluster. Sự thật là hai chữ số này cũng khá giống nhau. Điều tương tự xảy ra đối với hàng số 7 với các chữ số 7, 8, 9 được xếp vào 1 cluster. Với các cluster này, chúng ta có thể tiếp tục áp dụng K-means clustering để phân nhỏ cluster đó ra.

3. Trong clustering có một kỹ thuật thường được sử dụng là Hierarchical clustering (clustering phân tầng). Có hai loại Hierachical clustering:

- **Agglomerative** tức "đi từ dưới lên". Ban đầu coi mỗi điểm dữ liệu thuộc 1 cluster khác nhau, sau đó các cặp cluster gần giống nhau được gộp lại làm một cluster lớn hơn. 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.

5.2 Object Segmentation (tách vật thể trong ảnh)

5.2.1 Đặt vấn đề

Chúng ta cùng thử áp dụng thuật toán K-means clustering vào một bài toán xử lý ảnh khác: tách vật thể.

Giả sử chúng ta có bức ảnh dưới đây và muốn một thuật toán tự động nhận ra vùng khuôn mặt và tách nó ra.

5.2.2 Lên ý tưởng

(*Lại giả sử rằng chúng ta chưa biết gì khác ngoài K-means clustering, các bạn hãy dừng vài giây để nghĩ xem chúng ta có thể xử lý thế nào. Gợi ý: Có ba màu chủ đạo trong bức ảnh.*)

Ok, có ba màu, ba clusters!

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. Vậy chúng ta có thể áp dụng thuật toán K-means clustering để phân các pixel ảnh thành 3 clusters, sau đó chọn cluster chứa phần khuôn mặt (phần này do con người làm).

Đây là một bức ảnh màu, mỗi điểm ảnh sẽ được biểu diễn bởi 3 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 ta coi mỗi điểm dữ liệu là một vector 3 chiều chứa các giá trị này, sau đó áp dụng thuật toán K-means clustering, chúng ta có thể có kết quả mong muốn. Hãy thử xem

5.2.3 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]))
```

(*Phần còn lại của source code có thể xem tại đây*).

Sau khi tìm được các cluster, tôi thay giá trị của mỗi pixel bằng center của cluster chứa nó, và được kết quả như sau:

Ba màu: Hồng, đen, và màu da đã được phân nhóm. Và khuôn mặt có thể được tách ra từ phần có màu da (và vùng bên trong nó). Vậy là K-means clustering tạo ra một kết quả chấp nhận được.

5.3 Image Compression (nén ảnh và nén dữ liệu nói chung)

Để ý thấy rằng mỗi một pixel có thể nhận một trong số $256^3 = 16,777,216$ (16 triệu màu mà chúng ta vẫn nghe khi quảng cáo màn hình). 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, có cách nào không nếu ta chỉ biết K-means clustering?

Câu trả lời là có. Trong bài toán Segmentation phía trên, chúng ta có 3 clusters, và mỗi một điểm ảnh sau khi xử lý sẽ được biểu diễn bởi 1 số tương ứng với 1 cluster. Tuy nhiên, chất lượng bức ảnh rõ ràng đã giảm đi nhiều. Tôi làm một thí nghiệm nhỏ với số lượng clusters được tăng lên là 5, 10, 15, 20. Sau khi tìm được centers cho mỗi cluster, tôi thay giá trị của một điểm ảnh bằng giá trị của center tương ứng:

```
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 center
    for k in range(K):
        img4[label == k] = kmeans.cluster_centers_[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()
```

Kết quả:

Bạn đọc có thể quan sát là khi số lượng clusters tăng lên, chất lượng bức ảnh đã được cải thiện. Đồng thời, chúng ta chỉ cần lưu các centers và label của mỗi điểm ảnh là đã có được một bức ảnh nén (có mất dữ liệu).

5.4 Thảo luận

1. Với một thuật toán K-means clustering đơn giản, chúng ta đã có thể áp dụng nó vào các bài toán thực tế. Mặc dù kết quả chưa thực sự như ý, chúng ta vẫn thấy được tiềm năng của thuật toán đơn giản này.

2. Cách thay một điểm dữ liệu bằng center tương ứng là một trong số các kỹ thuật có tên chung là Vector Quantization (VQ). Không chỉ trong nén dữ liệu, VQ còn được áp dụng rộng rãi trong các thuật toán tạo Feature Vector cho các bài toán Phân loại (classification).

3. Các thuật toán trong bài toán này được áp dụng cho xử lý ảnh vì việc này giúp tôi dễ dàng minh họa kết quả hơn. Các kỹ thuật này hoàn toàn có thể áp dụng cho các loại cơ sở dữ liệu khác.

5.5 Souce code

Các bạn có thể download source code ở đây.

5.6 Tài liệu tham khảo

[1] Pattern Recognition and Machine Learning - Christopher Bishop

6

K-nearest neighbors

Nếu như con người có kiểu học "nước đến chân mới nhảy", thì trong Machine Learning cũng có một thuật toán như vậy.

6.1 Giới thiệu

6.1.1 Một câu chuyện vui

Có một anh bạn chuẩn bị đến ngày thi cuối kỳ. Vì môn này được mở tài liệu khi thi nên anh ta không chịu ôn tập để hiểu ý nghĩa của từng bài học và mối liên hệ giữa các bài. Thay vào đó, anh thu thập tất cả các tài liệu trên lớp, bao gồm ghi chép bài giảng (lecture notes), các slides và bài tập về nhà + lời giải. Để cho chắc, anh ta ra thư viện và các quán Photocopy quanh trường mua hết tất cả các loại tài liệu liên quan (*khá khen cho cậu này chịu khó tìm kiếm tài liệu*). Cuối cùng, anh bạn của chúng ta thu thập được một chồng cao tài liệu để mang vào phòng thi.

Vào ngày thi, anh tự tin mang chồng tài liệu vào phòng thi. Aha, đề này ít nhất mình phải được 8 điểm. Câu 1 giống hệt bài giảng trên lớp. Câu 2 giống hệt đề thi năm ngoái mà lời giải có trong tập tài liệu mua ở quán Photocopy. Câu 3 gần giống với bài tập về nhà. Câu 4 trắc nghiệm thậm chí cậu nhớ chính xác ba tài liệu có ghi đáp án. Câu cuối cùng, 1 câu khó nhưng anh đã từng nhìn thấy, chỉ là không nhớ ở đâu thôi.

Kết quả cuối cùng, cậu ta được 4 điểm, vừa đủ điểm qua môn. Cậu làm chính xác câu 1 vì tìm được ngay trong tập ghi chú bài giảng. Câu 2 cũng tìm được đáp án nhưng lời giải của quán Photocopy sai! Câu ba thấy gần giống bài về nhà, chỉ khác mỗi một số thôi, cậu cho kết quả giống như thế luôn, vậy mà không được điểm nào. Câu 4 thì tìm được cả 3 tài liệu nhưng có hai trong đó cho đáp án A, cái còn lại cho B. Cậu chọn A và được điểm. Câu 5 thì không làm được dù còn tới 20 phút, vì tìm mãi chẳng thấy đáp án đâu - nhiều tài liệu quá cũng mệt!!

Không phải ngẫu nhiên mà tôi dành ra ba đoạn văn để kể về chuyện học hành của anh chàng kia. Hôm nay tôi xin trình bày về một phương pháp trong Machine Learning, được gọi là

K-nearest neighbor (hay KNN), một thuật toán được xếp vào loại lazy (machine) learning (máy lười học). Thuật toán này khá giống với cách học/thi của anh bạn kém may mắn kia.

6.1.2 K-nearest neighbor

K-nearest neighbor là một trong những thuật toán supervised-learning đơn giản nhất (mà hiệu quả trong một vài trường hợp) trong Machine Learning. Khi training, thuật toán này *không học* một điều gì từ dữ liệu training (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 kết quả của dữ liệu mới. K-nearest neighbor có thể áp dụng được vào cả hai loại của bài toán Supervised learning là Classification và Regression. KNN còn được gọi là một thuật toán Instance-based hay Memory-based learning.

Có một vài khái niệm tương ứng người-máy như sau:

| Ngôn ngữ người | Ngôn ngữ Máy Học | in Machine Learning | |————————|————————|————————| | Câu hỏi | Điểm dữ liệu | Data point | | Đáp án | Đầu ra, nhãn | Output, Label | | Ôn thi | Huấn luyện | Training | | Tập tài liệu mang vào phòng thi | Tập dữ liệu tập huấn | Training set | | Đề thi | Tập dữ liểu kiểm thử | Test set | | Câu hỏi trong dề thi | Dữ liệu kiểm thử | Test data point | | Câu hỏi có đáp án sai | Nhiễu | Noise, Outlier | | Câu hỏi gần giống | Điểm dữ liệu gần nhất | Nearest Neighbor |

 Với KNN, trong bài toán Classification, label của một điểm dữ liệu mới (hay kết quả của câu hỏi trong bài thi) được suy ra trực tiếp từ K điểm dữ liệu gần nhất trong training set. Label của một test data có thể được quyết định bằng major voting (bầu chọn theo số phiếu) giữa các 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 label. Chi tiết sẽ được nêu trong phần tiếp theo.

Trong bài toán Regresssion, 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 khoảng cách tới các điểm gần nhất đó.

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 torng training set gần nó nhất (K-lân cận), *không quan tâm đến việc có một vài điểm dữ liệu trong những điểm gần nhất này là nhiễu*. Hình dưới đây là một ví dụ về KNN trong classification với K = 1.

Ví dụ trên đây là bài toán Classification với 3 classes: Đỏ, Lam, Lục. Mỗi điểm dữ liệu mới (test data point) sẽ được gán label theo màu của điểm mà nó thuộc về. Trong hình này, có một vài 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. Dẫn đến nếu dữ liệu test rơi vào vùng này sẽ có nhiều khả năng cho kết quả không chính xác.

6.1.3 Khoảng cách trong không gian vector

Trong không gian một chiều, khoảng cách giữa hai điểm là trị tuyệt đối giữa hiệu giá trị của hai điểm đó. Trong không gian nhiều chiều, khoảng cách giữa hai điểm có thể được định nghĩa bằng nhiều hàm số khác nhau, trong đó độ dài đường thẳng nối hai điểm chỉ là một trường hợp đặc biệt trong đó. Nhiều thông tin bổ ích (cho Machine Learning) có thể được tìm thấy tại Norms (chuẩn) của vector trong tab Math.

6.2 Phân tích toán học

Thuật toán KNN rất dễ hiểu nên sẽ phần "Phân tích toán học" này sẽ chỉ có 3 câu. Tôi trực tiếp đi vào các ví dụ. Có một điều đáng lưu ý là KNN phải *nhớ* tất cả các điểm dữ liệu training, việc này không được lợi về cả bộ nhớ và thời gian tính toán - giống như khi cậu bạn của chúng ta không tìm được câu trả lời cho câu hỏi cuối cùng.

6.3 Ví dụ trên Python

6.3.1 Bộ cơ sở dữ liệu Iris (Iris flower dataset).

Iris flower dataset là một bộ dữ liệu nhỏ (nhỏ hơn rất nhiều so với MNIST. Bộ dữ liệu này bao gồm thông tin của ba loại hoa Iris (một loài hoa lan) khác nhau: Iris setosa, Iris virginica và Iris versicolor. Mỗi loại có 50 bông hoa được đo với dữ liệu là 4 thông tin: chiều dài, chiều rộng đài hoa (sepal), và chiều dài, chiều rộng cánh hoa (petal). Dưới đây là ví dụ về hình ảnh của ba loại hoa. (Chú ý, đây không phải là bộ cơ sở dữ liệu ảnh như MNIST, mỗi điểm dữ liệu trong tập này chỉ là một vector 4 chiều).

Bộ dữ liệu nhỏ này thường được sử dụng trong nhiều thuật toán Machine Learning trong các lớp học. Tôi sẽ giải thích lý do không chọn MNIST vào phần sau.

6.3.2 Thí nghiệm

Trong phần này, chúng ta sẽ tách 150 dữ liệu trong Iris flower dataset ra thành 2 phần, gọi là *training set* và *test set*. Thuật toán KNN sẽ dựa vào thông tin ở *training set* để dự đoán xem mỗi dữ liệu trong *test set* tương ứng với loại hoa nào. Dữ liệu được dự đoán này sẽ được đối chiếu với loại hoa thật của mỗi dữ liệu trong *test set* 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.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors, datasets
```

Tiếp theo, chúng ta load dữ liệu và hiện thị vài dữ liệu mẫu. Các class được gán nhãn là 0, 1, và 2.

```
iris = datasets.load_iris()
iris_X = iris.data
iris_y = iris.target
print 'Number of classes: %d' %len(np.unique(iris_y))
print 'Number of data points: %d' %len(iris_y)


X0 = iris_X[iris_y == 0,:]
print '\nSamples from class 0:\n', X0[:5,:]

X1 = iris_X[iris_y == 1,:]
print '\nSamples from class 1:\n', X1[:5,:]

X2 = iris_X[iris_y == 2,:]
print '\nSamples from class 2:\n', X2[:5,:]
```

Number of classes: 3 Number of data points: 150

Samples from class 0: [[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2] [4.7 3.2 1.3 0.2] [4.6 3.1 1.5 0.2] [5. 3.6 1.4 0.2]]

Samples from class 1: [[7. 3.2 4.7 1.4] [6.4 3.2 4.5 1.5] [6.9 3.1 4.9 1.5] [5.5 2.3 4. 1.3] [6.5 2.8 4.6 1.5]]

Samples from class 2: [[6.3 3.3 6. 2.5] [5.8 2.7 5.1 1.9] [7.1 3. 5.9 2.1] [6.3 2.9 5.6 1.8] [6.5 3. 5.8 2.2]]

Nếu nhìn vào vài dữ liệu mẫu, chúng ta thấy rằng hai cột cuối mang khá nhiều thông tin giúp chúng ta có thể phân biệt được chúng. Chúng ta dự đoán rằng kết quả classification cho cơ sở dữ liệu này sẽ tương đối cao.

Tách training và test sets

Giả sử chúng ta muốn dùng 50 điểm dữ liệu cho test set, 100 điểm còn lại cho training set. Scikit-learn có một hàm số cho phép chúng ta ngẫu nhiên lựa chọn các điểm này, như sau:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
     iris_X, iris_y, test_size=50)

print "Training size: %d" %len(y_train)
print "Test size    : %d" %len(y_test)
```

Training size: 100 Test size : 50

Sau đây, tôi trước hết xét trường hợp đơn giản K = 1, tức là với mỗi điểm test data, ta chỉ xét 1 điểm training data gần nhất và lấy label của điểm đó để dự đoán cho điểm test này.

```
clf = neighbors.KNeighborsClassifier(n_neighbors = 1, p = 2)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print "Print results for 20 test data points:"
print "Predicted labels: ", y_pred[20:40]
print "Ground truth    : ", y_test[20:40]
```

Print results for first 20 test data points: Predicted labels: [2 1 2 2 1 2 2 0 2 0 2 0 1 0 0 2 2 0 2 0] Ground truth : [2 1 2 2 1 2 2 0 2 0 1 0 1 0 0 2 1 0 2 0]

 Kết quả cho thấy label dự đoán gần giống với label thật của test data, chỉ có 2 điểm trong số 20 điểm được hiển thị có kết quả sai lệch. Ở đây chúng ta làm quen với khái niệm mới: *ground truth*. Một cách đơn giản, *ground truth* chính là nhãn/label/đầu ra *thực sự* của các điểm trong test data. Khái niệm này được dùng nhiều trong Machine Learning, hy vọng lần tới các bạn gặp thì sẽ nhớ ngay nó là gì.

Phương pháp đánh giá (evaluation method)

Để đánh giá độ chính xác của thuật toán KNN classifier này, chúng ta xem xem có bao nhiêu điểm trong test data được dự đoán đúng. Lấy số lượng này chia cho tổng số lượng trong tập test data sẽ ra độ chính xác. Scikit-learn cung cấp hàm số `accuracy_score` để thực hiện công việc này.

```
from sklearn.metrics import accuracy_score
print "Accuracy of 1NN: %.2f %%" %(100*accuracy_score(y_test, y_pred))
```

Accuracy of 1NN: 94.00

1NN đã cho chúng ta kết quả là 94

Nhận thấy rằng chỉ xét 1 điểm gần nhất có thể dẫn đến kết quả sai nếu điểm đó là nhiễu. Một cách có thể làm tăng độ chính xác là tăng số lượng điểm lân cận lên, ví dụ 10 điểm, và xem xem trong 10 điểm gần nhất, class nào chiếm đa số thì dự đoán kết quả là class đó. Kỹ thuật dựa vào đa số này được gọi là major voting.

```
clf = neighbors.KNeighborsClassifier(n_neighbors = 10, p = 2)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print "Accuracy of 10NN with major voting: %.2f %%" %(100*accuracy_score(y_test,
    y_pred))
```

Accuracy of 10NN with major voting: 98.00

Kết quả đã tăng lên 98

Đánh trọng số cho các điểm lân cận

Là một kẻ tham lam, tôi chưa muốn dừng kết quả ở đây vì thấy rằng mình vẫn có thể cải thiện được. Trong kỹ thuật major voting bên trên, mỗi trong 10 điểm gần nhất được coi là có vai trò như nhau và giá trị *lá phiếu* của mỗi điểm này là như nhau. Tôi cho rằng như thế là không công bằng, vì rõ ràng rằng những điểm gần hơn nên có trọng số cao hơn (*càng thân cận thì càng tin tưởng*). Vậy nên tôi sẽ đánh trọng số khác nhau cho mỗi trong 10 điểm gần nhất này. Cách đánh trọng số phải thoải mãn điều kiện là một điểm càng gần điểm test data thì phải được đánh trọng số càng cao (tin tưởng hơn). Cách đơn giản nhất là lấy nghịch đảo của khoảng cách này. (Trong trường hợp test data trùng với 1 điểm dữ liệu trong training data, tức khoảng cách bằng 0, ta lấy luôn label của điểm training data).

Scikit-learn giúp chúng ta đơn giản hóa việc này bằng cách gán gía trị `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).

```
clf = neighbors.KNeighborsClassifier(n_neighbors = 10, p = 2, weights = 'distance')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print "Accuracy of 10NN (1/distance weights): %.2f %%" %(100*accuracy_score(y_test,
    y_pred))
```

Accuracy of 10NN (1/distance weights): 100.00

Aha, 100

Chú ý: 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 trong Machine Learning là:

$$w_i = \exp\left(\frac{-|||\mathbf{x} - \mathbf{x}_i|||_2^2}{\sigma^2}\right)$$

trong đó $\mathbf{x}$ là test data, $\mathbf{x}_i$ là một điểm trong K-lân cận của $\mathbf{x}$, w_i là trọng số của điểm đó (ứng với điểm dữ liệu đang xét $\mathbf{x}$), σ là một số dương. Nhận thấy rằng hàm số này cũng thỏa 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 = .5 # we can change this number
    return np.exp(-distances**2/sigma2)

clf = neighbors.KNeighborsClassifier(n_neighbors = 10, p = 2, weights = myweight)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print "Accuracy of 10NN (customized weights): %.2f %%" %(100*accuracy_score(y_test,
    y_pred))
```

Accuracy of 10NN (customized weights): 98.00

Trong trường hợp này, kết quả tương đương với kỹ thuật major voting. Để đánh giá chính xác hơn kết quả của KNN với K khác nhau, cách định nghĩa khoảng cách khác nhau và cách đánh trọng số khác nhau, chúng ta cần thực hiện quá trình trên với nhiều cách chia dữ liệu *training* và *test* khác nhau rồi lấy kết quả trung bình, vì rất có thể dữ liệu phân chia trong 1 trường hợp cụ thể là rất tốt hoặc rất xấu (bias). Đây cũng là cách thường được dùng khi đánh giá hiệu năng của một thuật toán cụ thể nào đó.

6.4 Thảo luận

6.4.1 KNN cho Regression

Với bài toán Regression, chúng ta cũng hoàn toàn có thể sử dụng phương pháp tương tự: ước lượng đầu ra dựa trên đầu ra và khoảng cách của các điểm trong K-lân cận. Việc ước lượng như thế nào các bạn có thể tự định nghĩa tùy vào từng bài toán.

6.4.2 Chuẩn hóa dữ liệu

Khi có một thuộc tính trong dữ liệu (hay phần tử trong vector) lớn hơn các thuộc tính khác rất nhiều (ví dụ thay vì đo bằng cm thì một kết quả lại tính bằng mm), khoảng cách giữa các điểm sẽ phụ thuộc vào thuộc tính này rất nhiều. Để có được kết quả chính xác hơn, một kỹ thuật thường được dùng là *Data Normalization* (chuẩn hóa dữ liệu) để đưa các thuộc tính có đơn vị đo khác nhau về cùng một khoảng giá trị, thường là từ 0 đến 1, trước khi thực hiện KNN. Có nhiều kỹ thuật chuẩn hóa khác nhau, các bạn sẽ được thấy khi tiếp tục theo dõi Blog này. Các kỹ thuật chuẩn hóa được áp dụng với không chỉ KNN mà còn với hầu hết các thuật toán khác.

6.4.3 Sử dụng các phép đo khoảng cách khác nhau

Ngoài norm 1 và norm 2 tôi giới thiệu trong bài này, còn rất nhiều các khoảng cách khác nhau có thể được dùng. Một ví dụ đơn giản là đếm số lượng thuộc tính khác nhau giữa hai điểm dữ liệu. Số này càng nhỏ thì hai điểm càng gần nhau. Đây chính là giả chuẩn 0 mà tôi đã giới thiệu trong Tab Math.

6.4.4 Ưu điểm của KNN

1. Độ phức tạp tính toán của quá trình training là bằng 0. 2. Việc dự đoán kết quả của dữ liệu mới rất đơn giản. 3. Không cần giả sử gì về phân phối của các class.

6.4.5 Nhược điểm của KNN

1. KNN rất nhạy cảm với nhiễu khi K nhỏ. 2. Như đã nói, KNN là một thuật toán mà mọi tính toán đều nằm ở khâu test. Trong đó việc tính khoảng cách tới *từng* điểm dữ liệu trong training set sẽ 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.

6.4.6 Tăng tốc cho KNN

Ngoài việc tính toán khoảng cách từ một điểm test data đến tất cả các điểm trong traing set (Brute Force), có một số thuật toán khác giúp tăng tốc việc tìm kiếm này. Bạn đọc có thể tìm kiếm thêm với hai từ khóa: K-D Tree và Ball Tree. Tôi xin dành phần này cho độc giả tự tìm hiểu, và sẽ quay lại nếu có dịp. Chúng ta vẫn còn những thuật toán quan trọng hơn khác cần nhiều sự quan tâm hơn.

6.4.7 Try this yourself

Tôi có viết một đoạn code ngắn để thực hiện việc Classification cho cơ sở dữ liệu MNIST. Các bạn hãy download toàn bộ bộ dữ liệu này về vì sau này chúng ta còn dùng nhiều, chạy thử, comment kết quả và nhận xét của các bạn vào phần comment bên dưới. Để trả lời cho câu hỏi vì sao tôi không chọn cơ sở dữ liệu này làm ví dụ, bạn đọc có thể tự tìm ra đáp án khi chạy xong đoạn code này.

Enjoy!

```
# %reset
import numpy as np
from mnist import MNIST # require `pip install python-mnist`
# https://pypi.python.org/pypi/python-mnist/

import matplotlib.pyplot as plt
from sklearn import neighbors
from sklearn.metrics import accuracy_score
import time

# you need to download the MNIST dataset first
# at: http://yann.lecun.com/exdb/mnist/
mndata = MNIST('../MNIST/') # path to your MNIST folder
mndata.load_testing()
mndata.load_training()
X_test = mndata.test_images
X_train = mndata.train_images
y_test = np.asarray(mndata.test_labels)
y_train = np.asarray(mndata.train_labels)


start_time = time.time()
```

```
clf = neighbors.KNeighborsClassifier(n_neighbors = 1, p = 2)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
end_time = time.time()
print "Accuracy of 1NN for MNIST: %.2f %%" %(100*accuracy_score(y_test, y_pred))
print "Running time: %.2f (s)" % (end_time - start_time)
```

6.4.8 Source code

iPython Notebook cho bài này có thể download tại đây.

6.5 Tài liệu tham khảo

1. sklearn.neighbors.NearestNeighbors

2. sklearn.model_selection.train_test_split

3. Tutorial To Implement k-Nearest Neighbors in Python From Scratch

7

Gradient Descent (phần 1/2)

7.1 Giới thiệu

Các bạn hẳn thấy hình vẽ dưới đây quen thuộc:

Điểm màu xanh lục là điểm local minimum (cực tiểu), và cũng là điểm làm cho hàm số đạt giá trị nhỏ nhất. Từ đây trở đi, tôi sẽ dùng *local minimum* để thay cho *điểm cực tiểu*, *global minimum* để thay cho *điểm mà tại đó hàm số đạt giá trị nhỏ nhất*. Global minimum là một trường hợp đặc biệt của local minimum.

Giả sử chúng ta đang quan tâm đến một hàm số một biến có đạo hàm mọi nơi. Xin cho tôi được nhắc lại vài điều đã quá quen thuộc:

1. Điểm local minimum x^* của hàm số là điểm có đạo hàm $f'(x^*)$ bằng 0. 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 1 điểm bất kỳ có hệ số góc chính bằng đạo hàm của hàm số tại điểm đó.

Trong hình phía trên, 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.

7.1.1 Gradient Descent

Trong Machine Learning nói riêng và Toán Tối Ưu nói chung, chúng ta thường xuyên phải tìm giá trị nhỏ nhất (hoặc đôi khi là lớn nhất) của một hàm số nào đó. Ví dụ như các hàm mất mát trong hai bài Linear Regression và K-means Clustering. Nhìn chung, việc tìm global minimum của các hàm mất mát trong Machine Learning là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. 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 (*đoạn này nghe rất quen thuộc, đúng không?*). Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng 0 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.

Hướng tiếp cận phổ biến nhất là xuất phát từ một điểm mà chúng ta 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 0. (Đây cũng chính là lý do phương pháp này được gọi là Gradient Descent - tức giảm *độ lớn* của đạo hàm). Gradient Descent (viết gọn là GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

Vì kiến thức về GD khá rộng nên tôi xin phép được chia thành hai phần. Phần 1 này giới thiệu ý tưởng phía sau thuật toán GD và một vài ví dụ đơn giản giúp các bạn làm quen với thuật toán này và vài khái niệm mới. Phần 2 sẽ nói về các phương pháp cải tiến GD và các biến thể của GD trong các bài toán mà số chiều và số điểm dữ liệu lớn. Những bài toán như vậy được gọi là *large-scale*.

7.2 Gradient Descent cho hàm 1 biến

Quay trở lại hình vẽ ban đầu và một vài quan sát tôi đã nêu. Giả sử x_t là điểm ta tìm được sau vòng lặp thứ t. Ta cần tìm một thuật toán để đưa x_t về càng gần x^* càng tốt.

Trong hình đầu tiên, chúng ta lại có thêm hai quan sát nữa:

1. Nếu đạo hàm của hàm số tại x_t: $f'(x_t) > 0$ 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, **chúng ta cần di chuyển ngược dấu với đạo hàm**:

$$x_{t+1} = x_t + \Delta$$

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 trực quan nhất, là tỉ lệ thuận với $-f'(x_t)$.

Hai nhận xét phía trên cho chúng ta một cách cập nhật đơn giản là:

$$x_{t+1} = x_t - \eta f'(x_t)$$

Trong đó η (đọc là *eta*) là một số dương được gọi là *learning rate* (tốc độ học). Dấu trừ thể hiện việc chúng ta phải đi ngược với đạo hàm. Các quan sát đơn giản phía trên, mặc dù không phải đúng cho tất cả các bài toán, là nên tảng cho rất nhiều phương pháp tối ưu nói chung và thuật toán Machine Learning nói riêng.

7.2.1 Ví dụ đơn giản với Python

Xét hàm số $f(x) = x^2 + 5\sin(x)$ với đạo hàm $f'(x) = 2x + 5\cos(x)$ (một lý do tôi chọn hàm này vì nó không dễ tìm nghiệm của đạo hàm bằng 0 như hàm phía trên). Giả sử bắt đầu từ một điểm x_0 nào đó, tại vòng lặp thứ t, chúng ta sẽ cập nhật như sau:

$$x_{t+1} = x_t - \eta(2x_t + 5\cos(x_t))$$

Như thường lệ, tôi khai báo vài thư viện quen thuộc

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import math
import numpy as np
import matplotlib.pyplot as plt
```

Tiếp theo, tôi viết các hàm số :

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 của 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 Gradient Desent nêu phía trên. Đầu vào của hàm số này là learning rate và điểm bắt đầu. 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(eta, x0):
    x = [x0]
    for it in range(100):
        x_new = x[-1] - eta*grad(x[-1])
        if abs(grad(x_new)) < 1e-3:
            break
        x.append(x_new)
    return (x, it)
```

Điểm khởi tạo khác nhau

Sau khi có các hàm cần thiết, tôi thử tìm nghiệm với các điểm khởi tạo khác nhau là $x_0 = -5$ và $x_0 = 5$.

```
(x1, it1) = myGD1(.1, -5)
(x2, it2) = myGD1(.1, 5)
print('Solution x1 = %f, cost = %f, obtained after %d iterations'%(x1[-1], cost(x1
    [-1]), it1))
print('Solution x2 = %f, cost = %f, obtained after %d iterations'%(x2[-1], cost(x2
    [-1]), it2))
```

Solution x1 = -1.110667, cost = -3.246394, obtained after 11 iterations Solution x2 = -1.110341, cost = -3.246394, obtained after 29 iterations

Vậy là với các điểm ban đầu khác nhau, thuật toán của chúng ta tìm được nghiệm gần giống nhau, mặc dù với tốc độ hội tụ khác nhau. Dưới đây là hình ảnh minh họa thuật toán GD cho bài toán này (*xem tốt trên Desktop ở chế độ full màn hình*).

Từ hình minh họa trên ta thấy rằng hình bên trái, tương ứng với $x_0 = -5$, nghiệm hội tụ nhanh hơn, vì điểm ban đầu x_0 gần với nghiệm $x\} \approx -1$ hơn. Hơn nữa, với $x_0 = 5$ ở hình bên phải, *đường đi* của nghiệm có chứa một khu vực có đạo hàm khá nhỏ gần điểm có hoành độ bằng 2. 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.

Learning rate khác nhau

Tốc độ hội tụ của GD không những phụ thuộc vào điểm khởi tạo ban đầu mà còn phụ thuộc vào *learning rate*. Dưới đây là một ví dụ với cùng điểm khởi tạo $x_0 = -5$ nhưng learning rate khác nhau:

Ta quan sát thấy hai điều:

1. Với *learning rate* nhỏ $\eta = 0.01$, tốc độ hội tụ rất chậm. Trong ví dụ này tôi chọn tối đa 100 vòng lặp nên thuật toán dừng lại trước khi tới *đích*, mặc dù đã rất gần. 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. 2. Với *learning rate* lớn $\eta = 0.5$, thuật toán tiến rất nhanh tới *gần đích* sau vài vòng lặp. Tuy nhiên, thuật toán không hội tụ được vì *bước nhảy* quá lớn, khiến nó cứ *quẩn quanh* ở đích.

Việc lựa chọn *learning rate* rất quan trọng trong các bài toán thực tế. Việc lựa chọn giá trị 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. Tôi sẽ quay lại vấn đề này ở phần 2.

7.3 Gradient Descent cho hàm nhiều biến

Giả sử ta cần tìm global minimum cho hàm $f(\theta)$ trong đó θ (*theta*) là một vector, thường được dùng để ký hiệu tập hợp các tham số của một mô hình cần tối ưu (trong Linear Regression thì các tham số chính là hệ số $\mathbf{w}$). Đạo hàm của hàm số đó tại một điểm θ bất kỳ được ký hiệu là $\nabla_\theta f(\theta)$ (hình tam giác ngược đọc là *nabla*). Tương tự như hàm 1 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:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

Hoặc viết dưới dạng đơn giản hơn: $\theta = \theta - \eta \nabla_\theta f(\theta)$.

Quy tắc cần nhớ: **luôn luôn đi ngược hướng với đạo hàm**.

Việc tính toán đạo hàm của các hàm nhiều biến là một kỹ năng cần thiết. Một vài đạo hàm đơn giản có thể được tìm thấy ở đây.

7.3.1 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.

Hàm mất mát của Linear Regression là:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \|\|\mathbf{y} - \bar{\mathbf{X}}\mathbf{w}\|\|_2^2$$

Chú ý: hàm này có khác một chút so với hàm tôi nêu trong bài Linear Regression. Mẫu số có thêm N là số lượng dữ liệu trong training set. Việc lấy trung bình cộng của lỗi này nhằm giúp tránh trường hợp hàm mất mát và đạo hàm có giá trị là một số rất lớn, ảnh hưởng tới độ chính xác của các phép toán khi thực hiện trên máy tính. Về mặt toán học, nghiệm của hai bài toán là như nhau.

Dạo hàm của hàm mất mát là:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{1}{N} \bar{\mathbf{X}}^T (\bar{\mathbf{X}}\mathbf{w} - \mathbf{y}) \qquad (1)$$

7.3.2 Sau đây là ví dụ trên Python và một vài lưu ý khi lập trình

Load thư viện

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
np.random.seed(2)
```

Tiếp theo, chúng ta tạo 1000 điểm dữ liệu được chọn *gần* với đường thẳng $y = 4 + 3x$, hiển thị chúng và tìm nghiệm theo công thức:

```
X = np.random.rand(1000, 1)
y = 4 + 3 * X + .2*np.random.randn(1000, 1)

# Building Xbar
one = np.ones((X.shape[0],1))
Xbar = np.concatenate((one, X), axis = 1)

A = np.dot(Xbar.T, Xbar)
b = np.dot(Xbar.T, y)
w_lr = np.dot(np.linalg.pinv(A), b)
print('Solution found by formula: w = ',w_lr.T)

# Display result
w = w_lr
w_0 = w[0][0]
w_1 = w[1][0]
x0 = np.linspace(0, 1, 2, endpoint=True)
y0 = w_0 + w_1*x0

# Draw the fitting line
plt.plot(X.T, y.T, 'b.')     # data
plt.plot(x0, y0, 'y', linewidth = 2)   # the fitting line
plt.axis([0, 1, 0, 10])
plt.show()
```

Solution found by formula: w = [[4.00305242 2.99862665]]

Đường thẳng tìm được là đường có màu vàng có phương trình $y \approx 4 + 2.998x$.

Tiếp theo ta viết đạo hàm và hàm mất mát:

```
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)**2;
```

Kiểm tra đạo hàm

Việc tính đạo hàm của hàm nhiều biến thông thường khá phức tạp và rất dễ mắc lỗi, nếu chúng ta tính sai đạo hàm thì thuật toán GD không thể chạy đúng được. 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 1 biến):

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

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

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

Cách tính này được gọi là *numerical gradient*.

Câu hỏi: Tại sao công thức xấp xỉ hai phía trên đây lại được sử dụng rộng rãi, sao không sử dụng công thức xấp xỉ đạo hàm bên phải hoặc bên trái?

Có hai các giải thích cho vấn đề này, một bằng hình học, một bằng giải tích.

Giải thích bằng hình học Quan sát hình dưới đây:

 Trong hình, vector màu đỏ là đạo hàm *chính xác* của hàm số tại điểm có hoành độ bằng x_0. Vector màu xanh lam (có vẻ là hơi tím sau khi convert từ .pdf sang .png) thể hiện cách xấp xỉ đạo hàm phía phải. Vector màu xanh lục thể hiện cách xấp xỉ đạo hàm 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.

Giải thích bằng giải tích Chúng ta cùng quay lại một chút với Giải tích I năm thứ nhất đại học: Khai triển Taylor.

Với ε rất nhỏ, ta có hai xấp xỉ sau:

$$f(x+\varepsilon) \approx f(x) + f'(x)\varepsilon + \frac{f"(x)}{2}\varepsilon^2 + \dots$$

và:

$$f(x-\varepsilon) \approx f(x) - f'(x)\varepsilon + \frac{f"(x)}{2}\varepsilon^2 - \dots$$

Từ đó ta có:

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

$$\frac{f(x+\varepsilon) - f(x-\varepsilon)}{2\varepsilon} \approx f'(x) + \frac{f^{(3)}(x)}{6}\varepsilon^2 + \dots = f'(x) + O(\varepsilon^2) \quad (4)$$

trong đó $O()$ là Big O notation.

Từ đó, nếu xấp xỉ đạo hàm bằng công thức (3) (xấp xỉ đạo hàm phải), sai số sẽ là $O(\varepsilon)$. Trong khi đó, nếu xấp xỉ đạo hàm bằng công thức (4) (xấp xỉ đạo hàm hai phía), sai số sẽ là $O(\varepsilon^2) \ll O(\varepsilon)$ nếu ε nhỏ.

Cả hai cách giải thích trên đây đều cho chúng ta thấy rằng, xấp xỉ đạo hàm hai phía là xấp xỉ tốt hơn.

Với hàm nhiều biến Với hàm nhiều biến, công thức (2) được áp dụng cho từng biến khi các biến khác cố định. Cách tính này 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. Khi so sánh đạo hàm này với đạo hàm chính xác tính theo công thức, người ta thường giảm số chiều dữ liệu và giảm số điểm dữ liệu để thuận tiện cho tính toán. Một khi đạo hàm tính được rất gần với *numerical gradient*, chúng ta có thể tự tin rằng đạo hàm tính được là chính xác.

Dưới đây là một đoạn code đơn giản để kiểm tra đạo hàm và có thể áp dụng với một hàm số (của một vector) bất kỳ với `cost` và `grad` đã tính ở phía trên.

```
def numerical_grad(w, cost):
    eps = 1e-4
    g = np.zeros_like(w)
    for i in range(len(w)):
        w_p = w.copy()
        w_n = w.copy()
        w_p[i] += eps
        w_n[i] -= eps
        g[i] = (cost(w_p) - cost(w_n))/(2*eps)
    return g

def check_grad(w, cost, grad):
    w = np.random.rand(w.shape[0], w.shape[1])
    grad1 = grad(w)
    grad2 = numerical_grad(w, cost)
    return True if np.linalg.norm(grad1 - grad2) < 1e-6 else False

print( 'Checking gradient...', check_grad(np.random.rand(2, 1), cost, grad))
```

```
Checking gradient... True
```

(*Với các hàm số khác, bạn đọc chỉ cần viết lại hàm* `grad` *và* `cost` *ở phần trên rồi áp dụng đoạn code này để kiểm tra đạo hàm. Nếu hàm số là hàm của một ma trận thì chúng ta thay đổi một chút trong hàm* `numerical_grad`*, tôi hy vọng không quá phức tạp*).

Với bài toán Linear Regression, cách tính đạo hàm như trong (1) phía trên được coi là đúng vì sai số giữa hai cách tính là rất nhỏ (nhỏ hơn 10^{-6}). Sau khi có được đạo hàm chính xác, chúng ta viết hàm cho GD:

```
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)

w_init = np.array([[2], [1]])
(w1, it1) = myGD(w_init, grad, 1)
```

```
print('Solution found by GD: w = ', w1[-1].T, ',\nafter %d iterations.' %(it1+1))
```

Solution found by GD: w = [[4.01780793 2.97133693]] , after 49 iterations.

Sau 49 vòng lặp, thuật toán đã hội tụ với một nghiệm khá gần với nghiệm tìm được theo công thức.

Dưới đây là hình động minh họa thuật toán GD.

Trong hình bên trái, các đường thẳng màu đỏ là nghiệm tìm được sau mỗi vòng lặp.

Trong hình bên phải, tôi xin giới thiệu một thuật ngữ mới: *đường đồng mức*.

Đường đồng mức (level sets)

Với đồ thị của một hàm số với hai biến đầu vào cần được vẽ trong không gian ba chiều, nhều khi chúng ta khó nhìn được nghiệm có khoảng tọa độ bao nhiêu. Trong toán tối ưu, người ta thường dùng một cách vẽ sử dụng khái niệm *đường đồng mức* (level sets).

Nếu các bạn để ý trong các bản độ tự nhiên, để miêu tả độ cao của các dãy núi, người ta dùng nhiều đường cong kín bao quanh nhau như sau:

 <img src ="http://files.vforum.vn/2016/T06/img/vforum.vn-324944-hinh-44-lc6b0e1bba3c-c491e1bb93-c491e1bb8ba-hc3acnh-te1bb89-le1bb87-le1bb9bn.png" align = "center" width = "600"> Ví dụ về đường đồng mức trong các bản đồ tự nhiên. (Nguồn: Địa lý 6: Đường đồng mức là những đường như thế nào?)

Các vòng nhỏ màu đỏ hơn thể hiện các điểm ở trên cao hơn.

Trong toán tối ưu, người ta cũng dùng phương pháp này để thể hiện các bề mặt trong không gian hai chiều.

Quay trở lại với hình minh họa thuật toán GD cho bài toán Liner Regression bên trên, hình bên phải là hình biểu diễn các level sets. Tức là tại các điểm trên cùng một vòng, hàm mất mát có giá trị như nhau. Trong ví dụ này, tôi hiển thị giá trị của hàm số tại một số vòng. Các vòng màu xanh có giá trị thấp, các vòng tròn màu đỏ phía ngoài có giá trị cao hơn. Điểm này khác một chút so với đường đồng mức trong tự nhiên là các vòng bên trong thường thể hiện một thung lũng hơn là một đỉnh núi (vì chúng ta đang đi tìm giá trị nhỏ nhất).

Tôi thử với *learning rate* nhỏ hơn, kết quả như sau:

Tốc độ hội tụ đã chậm đi nhiều, thậm chí sau 99 vòng lặp, GD vẫn chưa tới gần được nghiệm tốt nhất. Trong các bài toán thực tế, chúng ta cần nhiều vòng lặp hơn 99 rất nhiều, vì số chiều và số điểm dữ liệu thường là rất lớn.

7.4 Một ví dụ khác

Để kết thúc phần 1 của Gradient Descent, tôi xin nêu thêm một ví dụ khác.

Hàm số $f(x, y) = (x^2 + y - 7)^2 + (x - y + 1)^2$ có hai điểm local minimum màu xanh lục tại $(2, 3)$ và $(-3, -2)$, và chúng cũng là hai điểm global minimum. Trong ví dụ này, tùy vào điểm khởi tạo mà chúng ta thu được các nghiệm cuối cùng khác nhau.

7.5 Thảo luận

Dựa trên GD, có rất nhiều thuật toán phức tạp và hiệu quả hơn được thiết kế cho những loại bài toán khác nhau. Vì bài này đã đủ dài, tôi xin phép dừng lại ở đây. Mời các bạn đón đọc bài Gradient Descent phần 2 với nhiều kỹ thuật nâng cao hơn.

7.6 Tài liệu tham khảo

1. An overview of gradient descent optimization algorithms 2. http://www.benfrederickson.com/numerical-optimization/ 3. Gradient Descent by Andrew NG

8

Gradient Descent (phần 2/2)

Trong phần 1 của Gradient Descent (GD), tôi đã giới thiệu với bạn đọc về thuật toán Gradient Descent. Tôi xin nhắc lại rằng nghiệm cuối cùng của Gradient Descent phụ thuộc rất nhiều vào điểm khởi tạo và learning rate. Trong bài này, tôi xin đề cập một vài phương pháp thường được dùng để khắc phục những hạn chế của GD. Đồng thời, các thuật toán biến thể của GD thường được áp dụng trong các mô hình Deep Learning cũng sẽ được tổng hợp.

8.1 Các thuật toán tối ưu Gradient Descent

8.1.1 Momentum

Nhắc lại thuật toán Gradient Descent

Dành cho các bạn chưa đọc phần 1 của Gradient Descent. Để giải bài toán tìm điểm *global optimal* của hàm mất mát $J(\theta)$ (Hàm mất mát cũng thường được ký hiệu là $J()$ với θ là tập hợp các tham số của mô hình), tôi xin nhắc lại thuật toán GD:

———- **Thuật toán Gradient Descent:**

1. Dự đoán một điểm khởi tạo $\theta = \theta_0$. 2. Cập nhật θ đến khi đạt được kết quả chấp nhận được:

$$\theta = \theta - \eta \nabla_\theta J(\theta)$$

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 như hình một thung lũng giống như hình 1a) dưới đây. 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.

 <img src ="/assets/GD/momentum.png" align = "center" width = "800"> Hình 1: So sánh Gradient Descent với các hiện tượng vật lý

Tuy nhiên, nếu như bề mặt có hai đáy thung lũng như Hình 1b) thì tùy vào việc đặt bi ở A hay B, vị trí cuối cùng của bi sẽ ở C hoặc D. Điểm D là một điểm local minimum chúng ta không mong muốn.

Nếu suy nghĩ một cách vật lý hơn, vẫn trong Hình 1b), 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 xuống C như trong Hình 1c). Đây chính là điều chúng ta mong muốn. Bạn đọc có thể đặt câu hỏi rằng liệu bi lăn từ A tới C có theo *đà* lăn tới E rồi tới D không. Xin trả lời rằng điều này khó xảy ra hơn vì nếu so với dốc DE thì dốc CE cao hơn nhiều.

Dựa trên hiện tượng này, một thuật toán được ra đời nhằm khắc phục việc nghiệm của GD rơi vào một điểm local minimum không mong muốn. Thuật toán đó có tên là Momentum (tức *theo đà* trong tiếng Việt).

Gradient Descent với Momentum

Để biểu diễn *momentum* bằng toán học thì chúng ta phải làm thế nào?

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 $\theta_{t+1} = \theta_t - v_t$. Dấu trừ thể hiện việc phải di chuyển ngược với đạo hàm. Công việc của chúng ta bây giờ 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} (chúng ta coi như vận tốc ban đầu $v_0 = 0$). Một cách đơn giản nhất, ta có thể cộng (có trọng số) hai đại lượng này lại:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$

Trong đó γ thường được chọn là một giá trị khoảng 0.9, v_t 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 như sau:

$$\theta = \theta - v_t$$

Thuật toán đơn giản này tỏ ra rất hiệu quả trong các bài toán thực tế (trong không gian nhiều chiều, cách tính toán cũng hoàn tòan tương tự). Dưới đây là một ví dụ trong không gian một chiều.

Một ví dụ nhỏ

Chúng ta xem xét một hàm đơn giản có hai điểm local minimum, trong đó 1 điểm là global minimum:

$$f(x) = x^2 + 10\sin(x)$$

Có đạo hàm là: $f'(x) = 2x + 10\cos(x)$. Hình 2 dưới đây thể hiện sự khác nhau giữa thuật toán GD và thuật toán GD với Momentum:

 <table width = "100<tr > <td width="40<img style="display:block;" width = "100</td> <td width="40<img style="display:block;" width = "100</td> </tr> </table> Hình 2: Minh họa thuật toán GD với Momentum.

Hình bên trái là đường đi của nghiệm khi không sử dụng Momentum, thuật toán hội tụ sau chỉ 5 vòng lặp nhưng nghiệm tìm được là nghiệm local minimun.

Hình bên phải là đường đi của nghiệm khi có sử dụng Momentum, *hòn bi* đã có thể vượt dốc tới khu vực gần điểm global minimun, sau đó dao động xung quanh điểm này, giảm tốc rồi cuối cùng tới đích. Mặc dù mất nhiều vòng lặp hơn, GD với Momentum cho chúng ta nghiệm chính xác hơn. Quan sát đường đi của *hòn bi* trong trường hợp này, chúng ta thấy rằng điều này giống với vật lý hơn!

Nếu biết trước điểm *đặt bi* ban đầu `theta`, đạo hàm của hàm mất mát tại một điểm bất kỳ `grad(theta)`, lượng thông tin lưu trữ từ vần tốc trước đó `gamma` và learning rate `eta`, chúng ta có thể viết hàm số `GD_momentum` trong Python như sau:

```
# check convergence
def has_converged(theta_new, grad):
    return np.linalg.norm(grad(theta_new))/
                            len(theta_new) < 1e-3

def GD_momentum(theta_init, grad, 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 has_converged(theta_new, grad):
            break
        w.append(theta_new)
        v_old = v_new
    return theta
    # this variable includes all points in the path
    # if you just want the final answer,
```

```
    # use 'return theta[-1]'
```

8.1.2 Nesterov accelerated gradient (NAG)

Momentum giúp *hòn bi* vượt qua được *dốc locaminimum*, tuy nhiên, có một hạn chế chúng ta có thể thấy trong ví dụ trên: Khi tới gần *đích*, momemtum vẫn mất khá nhiều thời gian trước khi dừng lại. Lý do lại cũng chính là vì có *đà*. Có một phương pháp khác tiếp tục giúp khắc phục điều này, phương pháp đó mang tên Nesterov accelerated gradient (NAG), giúp cho thuật toán hội tụ nhanh hơn.

Ý tưởng chính

Ý tưởng cơ bản là *dự đoán hướng đi trong tương lai*, tức nhìn 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 hòn bi là $\theta - \gamma v_{t-1}$ (chúng ta không đính kèm phần gradient ở đây vì sẽ sử dụng nó trong bước cuối cùng). Vậy, thay vì sử dụng gradient của điểm hiện tại, NAG *đi trước một bước*, sử dụng gradient của điểm tiếp theo. Theo dõi hình dưới đây:

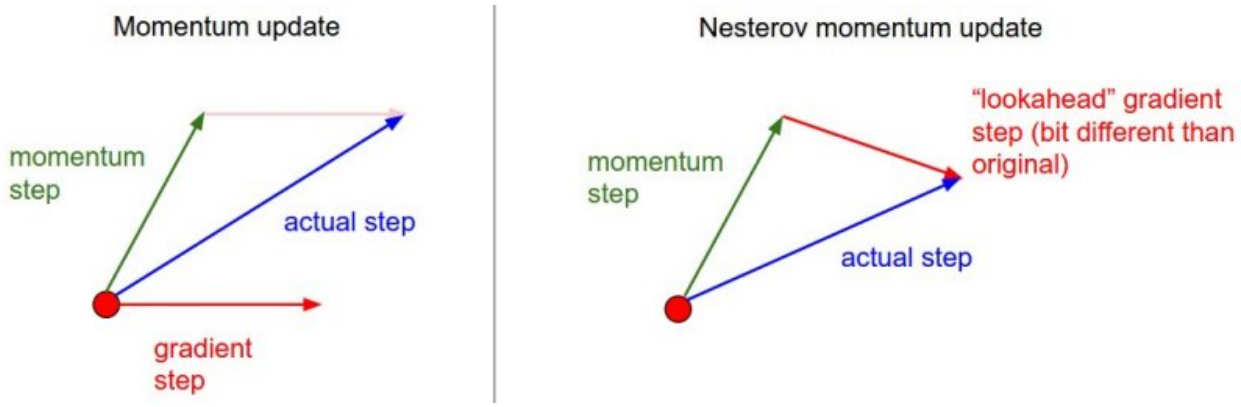

Fig. 8.1: Ý tưởng của Nesterov accelerated gradient. (Nguồn: CS231n Stanford: Convolutional Neural Networks for Visual Recognition

- Với momentum thông thường: *lượng thay đổi* là tổng của hai vector: momentum vector và gradient ở thời điểm hiện tại.
- Với Nesterove momentum: *lượng thay đổi* là tổng của hai vector: momentum vector và gradient ở thời điểm được xấp xỉ là điểm tiếp theo.

Công thức cập nhật

Công thức cập nhật của NAG được cho như sau:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

Để ý một chút các bạn sẽ thấy điểm được tính đạo hàm đã thay đổi.

Ví dụ minh họa

Dưới đây là ví dụ so sánh Momentum và NAG cho bài toán Linear Regression:

Hình bên trái là đường đi của nghiệm với phương pháp Momentum. nghiệm đi khá là *zigzag* và mất nhiều vòng lặp hơn. Hình bên phải là đường đi của nghiệm với phương pháp NAG, nghiệm hội tụ nhanh hơn, và đường đi ít *zigzag* hơn.

(Source code cho hình bên trái và hình bên phải).

8.1.3 Các thuật toán khác

Ngoài hai thuật toán trên, có rất nhiều thuật toán nâng cao khác được sử dụng trong các bài toán thực tế, đặc biệt là các bài toán Deep Learning. Có thể nêu một vài từ khóa như Adagrad, Adam, RMSprop,... Tôi sẽ không đề cập đến các thuật toán đó trong bài này mà sẽ dành thời gian nói tới nếu có dịp trong tương lai, khi blog đã đủ lớn và đã trang bị cho các bạn một lượng kiến thức nhất định.

Tuy nhiên, bạn đọc nào muốn đọc thêm có thể tìm được rất nhiều thông tin hữu ích trong bài này: An overview of gradient descent optimization algorithms .

8.2 Biến thể của Gradient Descent

Tôi xin một lần nữa dùng bài toán Linear Regression làm ví dụ. Hàm mất mát và đạo hàm của nó cho bài toán này lần lượt là (để cho thuận tiện, trong bài này tôi sẽ dùng ký hiệu $\mathbf{X}$ thay cho dữ liệu mở rộng $\bar{\mathbf{X}}$):

$$J(\mathbf{w}) = \frac{1}{2N}\|\|\mathbf{X}\mathbf{w} - \mathbf{y}\|\|_2^2$$
$$= \frac{1}{2N}\sum_{i=1}^{N}(\mathbf{x}_i\mathbf{w} - y_i)^2$$

và:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i^T(\mathbf{x}_i\mathbf{w} - y_i)$$

8.2.1 Batch Gradient Descent

Thuật toán Gradient Descent chúng ta nói từ đầu phần 1 đến giờ còn được gọi là Batch Gradient Descent. Batch ở đây được hiểu là *tất cả*, tức khi cập nhật $\theta = \mathbf{w}$, chúng ta sử dụng **tất cả** các điểm dữ liệu $\mathbf{x}_i$.

Cách làm này có một vài hạn chế đối với cơ sở dữ liệu có vô cùng nhiều điểm (hơn 1 tỉ người dùng của facebook chẳng hạn). Việc phải tính toán lại đạo hàm với tất cả các điểm này sau mỗi vòng lặp trở nên cồng kềnh và không hiệu quả. Thêm nữa, thuật toán này được coi là không hiệu quả với *online learning*.

Online learning là khi cơ sở dữ liệu được cập nhật liên tục (thêm người dùng đăng ký hàng ngày chẳng hạn), mỗi lần thêm vài điểm dữ liệu mới. Kéo theo đó là mô hình của chúng ta cũng phải thay đổi một chút để phù hợp với các dữ liệu mới này. Nếu làm theo Batch Gradient Descent, tức tính lại đạo hàm của hàm mất mát tại tất cả các điểm dữ liệu, thì thời gian tính toán sẽ rất lâu, và thuật toán của chúng ta coi như không *online* nữa do mất quá nhiều thời gian tính toán.

Trên thực tế, có một thuật toán đơn giản hơn và tỏ ra rất hiệu quả, có tên gọi là Stochastic Gradient Descent (SGD).

8.2.2 Stochastic Gradient Descent.

Trong thuật toán này, tại 1 thời điểm, 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. Việc này được thực hiện với từng điểm trên toàn bộ dữ liệu, sau đó lặp lại quá trình trên. Thuật toán rất đơn giản này trê thực tế lại làm việc rất hiệu quả.

Mỗi lần duyệt một lượt qua *tất cả* các điểm trên toàn bộ dữ liệu được gọi là một epoch. Với GD thông thường thì mỗi epoch ứng với 1 lần cập nhật θ, với SGD thì 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 1 epoch. Nhưng nhìn vào một mặt khác, SDG chỉ yêu cầu một lượng epoch rất nhỏ (thường là 10 cho lần đầu tiên, sau đó khi có dữ

liệu mới thì chỉ cần chạy dưới một epoch là đã có nghiệm tốt). Vì vậy SGD phù hợp với các bài toán có lượng cơ sở dữ liệu lớn (chủ yếu là Deep Learning mà chúng ta sẽ thấy trong phần sau của blog) và các bài toán yêu cầu mô hình thay đổi liên tục, tức online learning.

Thứ tự lựa chọn điểm dữ liệu

Một điểm cần lưu ý đó là: sau mỗi epoch, chúng ta cần shuffle (xáo trộn) thứ tự của các dữ liệu để đảm bảo tính ngẫu nhiên. Việc này cũng ảnh hưởng tới hiệu năng của SGD.

Một cách toán học, quy tắc cập nhật của SGD là:

$$\theta = \theta - \eta \nabla_\theta J(\theta; \mathbf{x}_i; \mathbf{y}_i)$$

trong đó $J(\theta; \mathbf{x}_i; \mathbf{y}_i)$ là hàm mất mát với chỉ 1 cặp điểm dữ liệu (input, label) là $(\mathbf{x}_i, \mathbf{y}_i)$. **Chú ý:** chúng ta hoàn toàn có thể áp dụng các thuật toán tăng tốc GD như Momentum, AdaGrad,... vào SGD.

Ví dụ với bài toán Linear Regression

Với bài toán Linear Regression, $\theta = \mathbf{w}$, hàm mất mát tại một điểm dữ liệu là:

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = \frac{1}{2}(\mathbf{x}_i \mathbf{w} - y_i)^2$$

Đạo hàm theo $\mathbf{w}$ tương ứng là:

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i; y_i) = \mathbf{x}_i^T (\mathbf{x}_i \mathbf{w} - y_i)^2$$

Và dưới đây là hàm số trong python để giải Linear Regression theo SGD:

```
# single point gradient
def sgrad(w, i, rd_id):
    true_i = rd_id[i]
    xi = Xbar[true_i, :]
    yi = y[true_i]
    a = np.dot(xi, w) - yi
    return (xi*a).reshape(2, 1)

def SGD(w_init, grad, eta):
    w = [w_init]
    w_last_check = w_init
    iter_check_w = 10
    N = X.shape[0]
    count = 0
    for it in range(10):
        # shuffle data
        rd_id = np.random.permutation(N)
        for i in range(N):
            count += 1
            g = sgrad(w[-1], i, rd_id)
```

```
            w_new = w[-1] - eta*g
            w.append(w_new)
            if count%iter_check_w == 0:
                w_this_check = w_new
                if np.linalg.norm(w_this_check - w_last_check)/len(w_init) < 1e-3:
                    return w
                w_last_check = w_this_check
    return w
```

Kết quả được cho như hình dưới đây (với dữ liệu được tạo giống như ở phần 1).

 <table width = "100<tr > <td width="40<img style="display:block;" width = "100</td> <td width="40<img style="display:block;" width = "100</td> </tr> </table> Trái: đường đi của nghiệm với SGD. Phải: giá trị của loss function tại 50 vòng lặp đầu tiên.

Hình bên trái mô tả đường đi của nghiệm. Chúng ta thấy rằng đường đi khá là *zigzag* chứ không *mượt* như khi sử dụng GD. Điều này là dễ hiểu vì một điểm dữ liệu không thể đại diện cho toàn bộ dữ liệu được. Tuy nhiên, chúng ta cũng thấy rằng thuật toán hội tụ khá nhanh đến vùng lân cận của nghiệm. Với 1000 điểm dữ liệu, SGD chỉ cần gần 3 epoches (2911 tương ứng với 2911 lần cập nhật, mỗi lần lấy 1 điểm). Nếu so với con số 49 vòng lặp (epoches) như kết quả tốt nhất có được bằng GD, thì kết quả này lợi hơn rất nhiều.

Hình bên phải mô tả hàm mất mát cho toàn bộ dữ liệu sau khi *chỉ* sử dụng 50 điểm dữ liệu đầu tiên. Mặc dù không *mượt*, tốc độ hội tụ vẫn rất nhanh.

Thực tế cho thấy chỉ lấy khoảng 10 điểm là ta đã có thể xác định được gần đúng phương trình đường thẳng cần tìm rồi. Đây chính là ưu điểm của SGD - hội tụ rất nhanh.

8.2.3 Mini-batch Gradient Descent

Khác với SGD, mini-batch sử dụng một số lượng n lớn hơn 1 (nhưng vẫn nhỏ hơn tổng số dữ liệu Nrất nhiều). Giống với SGD, Mini-batch Gradient Descent 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 n điểm dữ liệu (trừ mini-batch cuối có thể có ít hơn nếu N không chia hết cho n). Mỗi lần cập nhật, thuật toán này lấy ra một mini-batch để tính toán đạo hàm rồi cập nhật. Công thức có thể viết dưới dạng:

$$\theta = \theta - \eta \nabla_\theta J(\theta; \mathbf{x}_{i:i+n}; \mathbf{y}_{i:i+n})$$

Với $\mathbf{x}_{i:i+n}$ được hiểu là dữ liệu từ thứ i tới thứ $i+n-1$ (theo ký hiệu của Python). Dữ liệu này sau mỗi epoch là khác nhau vì chúng cần được xáo trộn. Một lần nữa, các thuật toán khác cho GD như Momentum, Adagrad, Adadelta,... cũng có thể được áp dụng vào đây.

Mini-batch GD được sử dụng trong hầu hết các thuật toán Machine Learning, đặc biệt là trong Deep Learning. Giá trị n thường được chọn là khoảng từ 50 đến 100.

Dưới đây là ví dụ về giá trị của hàm mất mát mỗi khi cập nhật tham số θ của một bài toán khác phức tạp hơn.

Để có thêm thông tin chi tiết hơn, bạn đọc có thể tìm trong bài viết rất tốt này.

8.3 Stopping Criteria (điều kiện dừng)

Có một điểm cũng quan trọng mà từ đầu tôi chưa nhắc đến: khi nào thì chúng ta biết thuật toán đã hội tụ và dừng lại?

Trong thực nghiệm, có một vài phương pháp như dưới đây:

1. Giới hạn số vòng lặp: đây là phương pháp phổ biến nhất và cũng để đảm bảo rằng chương trình chạy không quá lâu. Tuy nhiên, một nhược điểm của cách làm này là có thể thuật toán dừng lại trước khi đủ gần với nghiệm. 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 (ví dụ như khi có quá nhiều dữ liệu), nếu áp dụng phương pháp này thì coi như ta không được lợi khi sử dụng SGD và mini-batch GD. 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 *bẳng phẳng* tại một khu vực nhưng khu vực đó không chứa điểm local minimum (khu vực này thường được gọi là saddle points), thuật toán cũng dừng lại trước khi đạt giá trị mong muốn. 4. Trong SGD và mini-batch GD, cách thường dùng là so sánh nghiệm sau một vài lần cập nhật. Trong đoạn code Python phía trên về SGD, tôi áp dụng việc so sánh này mỗi khi nghiệm được cập nhật 10 lần. Việc làm này cũng tỏ ra khá hiệu quả.

8.4 Một phương pháp tối ưu đơn giản khác: Newton's method

Nhân tiện đang nói về tối ưu, tôi xin giới thiệu một phương pháp nữa có cách giải thích đơn giản: Newton's method. Các phương pháp GD tôi đã trình bày còn được gọi là first-order methods, vì lời giải tìm được dựa trên đạo hàm bậc nhất của hàm số. Newton's method là một second-order method, tức lời giải yêu cầu tính đến đạo hàm bậc hai.

Nhắc lại rằng, cho tới thời điểm này, chúng ta luôn giải phương trình đạo hàm của hàm mất mát bằng 0 để tìm các điểm local minimun. (Và trong nhiều trường hợp, coi nghiệm tìm được là nghiệm của bài toán tìm giá trị nhỏ nhất của hàm mất mát). Có một thuật toán nổi tiếng giúp giải bài toán $f(x) = 0$, thuật toán đó có tên là Newton's method.

8.4.1 Newton's method cho giải phương trình $f(x) = 0$

Thuật toán Newton's method được mô tả trong hình động minh họa dưới đây:

Ý tưởng giải bài toán $f(x) = 0$ bằng phương pháp Newton's method như sau. Xuất phát từ một điểm x_0 được cho là gần với nghiệm x^*. Sau đó vẽ đường tiếp tuyến (mặt tiếp tuyến trong không gian nhiều chiều) với đồ thị hàm số $y = f(x)$ tại điểm trên đồ thị có hoành độ x_0. Giao điểm x_1 của đường tiếp tuyến này với trục hoành được xem là gần với nghiệm x^* hơn. Thuật toán lặp lại với điểm mới x_1 và cứ như vậy đến khi ta được $f(x_t) \approx 0$.

Đó là ý nghĩa hình học của Newton's method, chúng ta cần một công thức để có thể dựa vào đó để lập trình. Việc này không quá phức tạp với các bạn thi đại học môn toán ở VN. Thật vậy, phương trình tiếp tuyến với đồ thị của hàm $f(x)$ tại điểm có hoành độ x_t là:

$$y = f'(x_t)(x - x_t) + f(x_t)$$

Giao điểm của đường thẳng này với trục x tìm được bằng cách giải phương trình vế phải của biểu thức trên bằng 0, tức là:

$$x = x_t - \frac{f(x_t)}{f'(x_t)} \triangleq x_{t+1}$$

8.4.2 Newton's method trong bài toán tìm local minimun

Áp dụng phương pháp này cho việc giải phương trình $f'(x) = 0$ ta có:

$$x_{t+1} = x_t - (f"(x_t))^{-1} f'(x_t)$$

Và trong không gian nhiều chiều với θ là biến:

$$\theta = \theta - \mathbf{H}(J(\theta))^{-1} \nabla_\theta J(\theta)$$

trong đó $\mathbf{H}(J(\theta))$ là đạo hàm bậc hai của hàm mất mất (còn gọi là Hessian matrix). Biểu thức này là một ma trận nếu θ là một vector. Và $\mathbf{H}(J(\theta))^{-1}$ chính là nghịch đảo của ma trận đó.

8.4.3 Hạn chế của Newton's method

- Điểm khởi tạo phải *rất* gần với nghiệm x^*. Ý tưởng sâu xa hơn của Newton's method là dựa trên khai triển Taylor của hàm số $f(x)$ tới đạo hàm thứ nhất:

  $$0 = f(x^\}) \approx f(x_t) + f'(x_t)(x_t - x^\})$$

  Từ đó suy ra: $x^\} \approx x_t - \frac{f(x_t)}{f'(x_t)}$. Một điểm rất quan trọng, khai triển Taylor chỉ đúng nếu x_t rất gần với x^*! Dưới đây là một ví dụ kinh điển trên Wikipedia về việc Newton's method cho một dãy số phân kỳ (divergence).

- Nhận thấy rằng trong việc giải phương trình $f(x) = 0$, chúng ta có đạo hàm ở mẫu số. Khi đạo hàm này gần với 0, ta sẽ được một đường thẳng song song hoặc gần song song với trục hoành. Ta sẽ hoặc không tìm được giao điểm, hoặc được một giao điểm ở vô cùng. Đặc biệt, khi nghiệm chính là điểm có đạo hàm bằng 0, thuật toán gần như sẽ không tìm được nghiệm!

- Khi áp dụng Newton's method cho bài toán tối ưu trong không gian nhiều chiều, chúng ta cần tính nghịch đảo của Hessian matrix. Khi số chiều và số điểm dữ liệu lớn, đạo hàm bậc hai của hàm mất mát sẽ là một ma trận rất lớn, ảnh hưởng tới cả memory và tốc độ tính toán của hệ thống.

8.5 Kết luận

Qua hai bài viết về Gradient Descent này, tôi hy vọng các bạn đã hiểu và làm quen với một thuật toán tối ưu được sử dụng nhiều nhất trong Machine Learning và đặc biệt là Deep Learning. Còn nhiều biến thể khác khá thú vị về GD (mà rất có thể tôi chưa biết tới), nhưng tôi xin phép được dừng chuỗi bài về GD tại đây và tiếp tục chuyển sang các thuật toán thú vị khác.

Hy vọng bài viết có ích với các bạn.

8.6 Tài liệu tham khảo

[1] Newton's method - Wikipedia

[2] An overview of gradient descent optimization algorithms

[3] Stochastic Gradient Descent - Wikipedia

[4] Stochastic Gradient Descen - Andrew Ng

[5] Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543– 547.

9

Perceptron Learning Algorithm

—

Cứ làm đi, sai đâu sửa đấy, cuối cùng sẽ thành công!

Đó chính là ý tưởng chính của một thuật toán rất quan trọng trong Machine Learning - thuật toán Perceptron Learning Algorithm hay PLA.

9.1 Giới thiệu

Trong bài này, tôi sẽ giới thiệu thuật toán đầu tiên trong Classification có tên là Perceptron Learning Algorithm (PLA) hoặc đôi khi được viết gọn là Perceptron.

Perceptron là một thuật toán Classification cho trường hợp đơn giản nhất: chỉ có hai class (lớp) (*bài toán với chỉ hai class được gọi là binary classification*) và cũng chỉ hoạt động được trong một trường hợp rất cụ thể. Tuy nhiên, nó là nền tảng cho một mảng lớn quan trọng của Machine Learning là Neural Networks và sau này là Deep Learning. (Tại sao lại gọi là Neural Networks - tức mạng dây thần kinh - các bạn sẽ được thấy ở cuối bài).

Giả sử chúng ta có hai tập hợp dữ liệu đã được gán nhãn được minh hoạ trong Hình 1 bên trái dưới đây. Hai class của chúng ta là 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 *classifier* (bộ phân lớp) để khi có một điểm dữ liệu hình tam giác màu xám mới, ta có thể dự đoán được màu (nhãn) của nó.

<table width = "100<tr > <td width="40<img style="display:block;" width = "100</td> <td width="40<img style="display:block;" width = "100</td> </tr> </table> Hình 1: Bài toán Perceptron

Hiểu theo một cách khác, chúng ta cần tìm *lãnh thổ* của mỗi class sao cho, với mỗi một điểm mới, ta chỉ cần xác định xem nó nằm vào lãnh thổ của class nào rồi quyết định nó thuộc class đó. Để tìm *lãnh thổ* của mỗi class, chúng ta cần đi tìm biên giới (boundary)

giữa hai *lãnh thổ* này. Vậy bài toán classification có thể coi là bài toán đi tìm boundary giữa các class. Và boundary 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 là một siêu mặt phẳng (hyperplane) (tôi gọi chung những boundary này là *đường phẳng*). Những boundary phẳng này được coi là đơn giản vì nó có thể biểu diễn dưới dạng toán học bằng một hàm số đơn giản có dạng tuyến tính, tức linear. Tất nhiên, chúng ta đang giả sử rằng tồn tại một đường phẳng để có thể phân định *lãnh thổ* của hai class. Hình 1 bên phải minh họa một đường thẳng phân chia hai class trong mặt phẳng. Phần có nền màu xanh được coi là *lãnh thổ* của lớp xanh, phần có nên màu đỏ được coi là *lãnh thổ* của lớp đỏ. Trong trường hợp này, điểm dữ liệu mới hình tam giác được phân vào class đỏ.

9.1.1 Bài toán Perceptron

Bài toán Perceptron được phát biểu như sau: *Cho hai class được gán nhãn, hãy tìm một đường phẳng sao cho toàn bộ các điểm thuộc class 1 nằm về 1 phía, toàn bộ các điểm thuộc class 2 nằm về phía còn lại của đường phẳng đó. Với giả định rằng tồn tại một đường phẳng như thế.*

Nếu tồn tại một đường phẳng phân chia hai class thì ta gọi hai class đó là *linearly separable*. Các thuật toán classification tạo ra các boundary là các đường phẳng được gọi chung là Linear Classifier.

9.2 Thuật toán Perceptron (PLA)

Cũng giống như các thuật toán lặp trong K-means Clustering và Gradient Descent, ý tưởng cơ bản của PLA là xuất phát từ một nghiệm dự đoán nào đó, qua mỗi vòng lặp, nghiệm sẽ được cập nhật tới một ví trí tốt hơn. Việc cập nhật này dựa trên việc giảm giá trị của một hàm mất mát nào đó.

9.2.1 Một số ký hiệu

Giả sử $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N] \in \mathbb{R}^{d \times N}$ là ma trận chứa các điểm dữ liệu mà mỗi cột $\mathbf{x}_i \in \mathbb{R}^{d \times 1}$ là một điểm dữ liệu trong không gian d chiều. (*Chú ý: khác với các bài trước tôi thường dùng các vector hàng để mô tả dữ liệu, trong bài này tôi dùng vector cột để biểu diễn. Việc biểu diễn dữ liệu ở dạng hàng hay cột tùy thuộc vào từng bài toán, miễn sao cách biểu diễn toán học của nó khiến cho người đọc thấy dễ hiểu*).

Giả sử thêm các nhãn tương ứng với từng điểm dữ liệu được lưu trong một vector hàng $\mathbf{y} = [y_1, y_2, \ldots, y_N] \in \mathbb{R}^{1 \times N}$, với $y_i = 1$ nếu $\mathbf{x}_i$ thuộc class 1 (xanh) và $y_i = -1$ nếu $\mathbf{x}_i$ thuộc class 2 (đỏ).

Tại một thời điểm, giả sử ta tìm được boundary là đường phẳng có phương trình:

$$f_{\mathbf{w}}(\mathbf{x}) = w_1 x_1 + \cdots + w_d x_d + w_0 \tag{9.1}$$
$$= \mathbf{w}^T \bar{\mathbf{x}} = 0 \tag{9.2}$$

với $\bar{\mathbf{x}}$ là điểm dữ liệu mở rộng bằng cách thêm phần tử $x_0 = 1$ lên trước vector $\mathbf{x}$ tương tự như trong Linear Regression. Và từ đây, khi nói $\mathbf{x}$, tôi cũng ngầm hiểu là điểm dữ liệu mở rộng.

Để cho đơn giản, chúng ta hãy cùng làm việc với trường hợp mỗi điểm dữ liệu có số chiều $d = 2$. Giả sử đường thẳng $w_1 x_1 + w_2 x_2 + w_0 = 0$ chính là nghiệm cần tìm như Hình 2 dưới đây:

Nhận xét rằng các điểm nằm về cùng 1 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 class. 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 class của nó bằng phép toán đơn giản như sau:

$$\text{label}(\mathbf{x}) = 1 \text{ if } \mathbf{w}^T \mathbf{x} \geq 0, \text{otherwise} - 1$$

Ngắn gọn hơn:

$$\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x})$$

trong đó, sgn là hàm xác định dấu, với giả sử rằng $\text{sgn}(0) = 1$.

9.2.2 Xây dựng hàm mất mát

Tiếp theo, chúng ta cần xây dựng hàm mất mát với tham số $\mathbf{w}$ bất kỳ. Vẫn trong không gian hai chiều, giả sử đường thẳng $w_1 x_1 + w_2 x_2 + w_0 = 0$ được cho như Hình 3 dưới đây:

Trong trường hợp này, các điểm được khoanh tròn là các điểm bị misclassified (phân lớp lỗi). Điều chúng ta mong muốn là không có điểm nào bị misclassified. Hàm mất mát đơn giản nhất chúng ta nghĩ đến là hàm *đếm* số lượng các điểm bị misclassied và tìm cách tối thiểu hàm số này:

$$J_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \text{sgn}(\mathbf{w}^T \mathbf{x_i}))$$

trong đó $\mathcal{M}$ là tập hợp các điểm bị misclassifed (*tập hợp này thay đổi theo* $\mathbf{w}$). Với mỗi điểm $\mathbf{x}_i \in \mathcal{M}$, vì điểm này bị misclassified nên y_i và $\text{sgn}(\mathbf{w}^T \mathbf{x})$ khác nhau, và vì thế $-y_i \text{sgn}(\mathbf{w}^T \mathbf{x_i}) = 1$. Vậy $J_1(\mathbf{w})$ chính là hàm *đếm* số lượng các điểm bị misclassified. Khi hàm số này đạt giá trị nhỏ nhất bằng 0 thì ta không còn điểm nào bị misclassified.

Một điểm quan trọng, hàm số này là rời rạc, không tính được đạo hàm theo $\mathbf{w}$ nên rất khó tối ưu. Chúng ta cần tìm một hàm mất mát khác để việc tối ưu khả thi hơn.

Xét hàm mất mát sau đây:

$$J(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T \mathbf{x_i})$$

Hàm $J()$ khác một chút với hàm $J_1()$ ở việc bỏ đi hàm sgn. Nhận xét rằng khi một điểm misclassified $\mathbf{x}_i$ nằm càng xa boundary thì giá trị $-y_i \mathbf{w}^T \mathbf{x}_i$ sẽ càng lớn, nghĩa là sự sai lệch càng lớn. Giá trị nhỏ nhất của hàm mất mát này cũng bằng 0 nếu không có điểm nào bị misclassifed. Hàm mất mát này cũng được cho là tốt hơn hàm $J_1()$ vì hàm này *trừng phạt* rất nặng những điểm *lấn sâu sang lãnh thổ của class kia*. Trong khi đó, $J()$ *trừng phạt* các điểm misclassified như nhau (đều = 1), bất kể chúng xa hay gần với đường biên giới.

Tại một thời điểm, nếu chúng ta chỉ quan tâm tới các điểm bị misclassified thì hàm số $J(\mathbf{w})$ khả vi (tính được đạo hàm), vậy chúng ta có thể sử dụng Gradient Descent hoặc Stochastic Gradient Descent (SGD) để tối ưu hàm mất mát này. Với ưu điểm của SGD cho các bài toán large-scale, chúng ta sẽ làm theo thuật toán này.

Với *một* điểm dữ liệu $\mathbf{x}_i$ bị misclassified, hàm mất mát trở thành:

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{w}^T \mathbf{x}_i$$

Đạo hàm tương ứng:

$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i \mathbf{x}_i$$

Vậy quy tắc cập nhật là:

$$\mathbf{w} = \mathbf{w} + \eta y_i \mathbf{x}_i$$

với η là learning rate.

Nhận xét rằng nếu $\mathbf{w}$ là nghiệm thì $\eta \mathbf{w}$ cũng là nghiệm với η là một số khác 0 bất kỳ. Vậy nếu $\mathbf{w}_0$ nhỏ gần với 0 và số vòng lặp đủ lớn, ta có thể coi như learning rate $\eta = 1$. Ta có một quy tắc cập nhật rất gọn là: $\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$. Nói cách khác, với mỗi điểm $\mathbf{x}_i$ bị misclassifed, ta chỉ cần nhân điểm đó với nhãn y_i của nó, lấy kết quả cộng vào $\mathbf{w}$ ta sẽ được $\mathbf{w}$ mới.

Ta có một quan sát nhỏ ở đây:

$$\mathbf{w}_{t+1}^T \mathbf{x}_i = (\mathbf{w}_t + y_i \mathbf{x}_i)^T \mathbf{x}_i \ = \mathbf{w}_t^T \mathbf{x}_i + y_i \|\|\mathbf{x}_i\|\|_2^2$$

Nếu $y_i = 1$, vì $\mathbf{x}_i$ bị misclassifed 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 $x_0 = 1$), nghĩa là $\mathbf{w}_{t+1}^T \mathbf{x}_i > \mathbf{w}_t^T \mathbf{x}_i$. Lý giải bằng lời, $\mathbf{w}_{t+1}$ tiến về phía làm cho $\mathbf{x}_i$ được phân lớp đúng. Điều tương tự xảy ra nếu $y_i = -1$.

Đến đây, cảm nhận của chúng ta với thuật toán này là: cứ chọn đường boundary đi. Xét từng điểm một, nếu điểm đó bị misclassified thì tiến đường boundary về phía làm cho điểm đó được classifed đúng. Có thể thấy rằng, khi di chuyển đường boundary này, các điểm trước đó được classified đúng có thể lại bị misclassified. Mặc dù vậy, PLA vẫn được đảm bảo sẽ hội tụ sau một số hữu hạn bước (tôi sẽ chứng minh việc này ở phía sau của bài viết). Tức là cuối cùng, ta sẽ tìm được đường phẳng phân chia hai lớp, miễn là hai lớp đó là linearly separable. Đây cũng chính là lý do câu đầu tiên trong bài này tôi nói với các bạn là: "Cứ làm đi, sai đâu sửa đấy, cuối cùng sẽ thành công!".

Tóm lại, thuật toán Perceptron có thể được viết như sau:

9.2.3 Tóm tắt PLA

1. Chọn ngẫu nhiên một vector hệ số $\mathbf{w}$ với các phần tử gần 0. 2. Duyệt ngẫu nhiên qua từng điểm dữ liệu $\mathbf{x}_i$:

- Nếu $\mathbf{x}_i$ được phân lớp đúng, tức $\text{sgn}(\mathbf{w}^T\mathbf{x}) = y_i$, chúng ta không cần làm gì.
- Nếu $\mathbf{x}_i$ bị misclassifed, cập nhật $\mathbf{w}$ theo công thức:

$$\mathbf{w} = \mathbf{w} + y_i\mathbf{x}_i$$

3. Kiểm tra xem có bao nhiêu điểm bị misclassifed. Nếu không còn điểm nào, dừng thuật toán. Nếu còn, quay lại bước 2.

9.3 Ví dụ trên Python

Như thường lệ, chúng ta sẽ thử một ví dụ nhỏ với Python.

9.3.1 Load thư viện và tạo dữ liệu

Chúng ta sẽ tạo hai nhóm dữ liệu, mỗi nhóm có 10 điểm, mỗi điểm dữ liệu có hai chiều để thuận tiện cho việc minh họa. Sau đó, tạo dữ liệu mở rộng bằng cách thêm 1 vào đầu mỗi điểm dữ liệu.

```
# generate data
# list of points
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
np.random.seed(2)

means = [[2, 2], [4, 2]]
```

```
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N).T
X1 = np.random.multivariate_normal(means[1], cov, N).T

X = np.concatenate((X0, X1), axis = 1)
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1)
# Xbar
X = np.concatenate((np.ones((1, 2*N)), X), axis = 0)
```

Sau khi thực hiện đoạn code này, biến `X` sẽ chứa dữ liệu input (mở rộng), biến `y` sẽ chứa nhãn của mỗi điểm dữ liệu trong `X`.

9.3.2 Các hàm số cho PLA

Tiếp theo chúng ta cần viết 3 hàm số cho PLA:

1. `h(w, x)`: tính đầu ra khi biết đầu vào `x` và weights `w`. 2. `has_converged(X, y, w)`: kiểm tra xem thuật toán đã hội tụ chưa. Ta chỉ cần so sánh `h(w, X)` với *ground truth* `y`. Nếu giống nhau thì dừng thuật toán. 3. `perceptron(X, y, w_init)`: hàm chính thực hiện PLA.

```
def h(w, x):
    return np.sign(np.dot(w.T, x))

def has_converged(X, y, w):
    return np.array_equal(h(w, X), y)

def perceptron(X, y, w_init):
    w = [w_init]
    N = X.shape[1]
    d = X.shape[0]
    mis_points = []
    while True:
        # mix data
        mix_id = np.random.permutation(N)
        for i in range(N):
            xi = X[:, mix_id[i]].reshape(d, 1)
            yi = y[0, mix_id[i]]
            if h(w[-1], xi)[0] != yi: # misclassified point
                mis_points.append(mix_id[i])
                w_new = w[-1] + yi*xi
                w.append(w_new)

        if has_converged(X, y, w[-1]):
            break
    return (w, mis_points)

d = X.shape[0]
w_init = np.random.randn(d, 1)
(w, m) = perceptron(X, y, w_init)
```

Dưới đây là hình minh họa thuật toán PLA cho bài toán nhỏ này:

Sau khi cập nhật 18 lần, PLA đã hội tụ. Điểm được khoanh tròn màu đen là điểm misclassified tương ứng được chọn để cập nhật đường boundary.

Source code cho phần này (bao gồm hình động) có thể được tìm thấy ở đây.

9.4 Chứng minh hội tụ

Giả sử rằng $\mathbf{w}^*$ là một nghiệm của bài toán (ta có thể giả sử việc này được vì chúng ta đã có giả thiết hai class là linearly separable - tức tồn tại nghiệm). Có thể thấy rằng, với mọi $\alpha > 0$, nếu $\mathbf{w}^*$ là nghiệm, $\alpha\mathbf{w}^*$ cũng là nghiệm của bài toán. Xét dãy số không âm $u_\alpha(t) = \|\|\mathbf{w}_t - \alpha\mathbf{w}^*\|\|_2^2$. Với $\mathbf{x}_i$ là một điểm bị misclassified nếu dùng nghiệm $\mathbf{w}_t$ ta có:

$$
\begin{aligned}
u_\alpha(t+1) &= \|\|\mathbf{w}_{t+1} - \alpha\mathbf{w}^*\|\|_2^2 && (9.3)\\
&= \|\|\mathbf{w}_t + y_i\mathbf{x}_i - \alpha\mathbf{w}^*\|\|_2^2 && (9.4)\\
&= \|\|\mathbf{w}_t - \alpha\mathbf{w}^*\|\|_2^2 + y_i^2\|\|\mathbf{x}_i\|\|_2^2 + 2y_i\mathbf{x}_i^T(\mathbf{w} - \alpha\mathbf{w}^*) && (9.5)\\
&< u_\alpha(t) \ + \|\|\mathbf{x}_i\|\|_2^2 - 2\alpha y_i\mathbf{x}_i^T\mathbf{w}\} && (9.6)
\end{aligned}
$$

Dấu nhỏ hơn ở dòng cuối là vì $y_i^2 = 1$ và $2y_i\mathbf{x}_i^T\mathbf{w}_t < 0$. Nếu ta đặt:

$$
\begin{aligned}
\beta^2 &= \max_{i=1,2,\ldots,N} \|\|\mathbf{x}_i\|\|_2^2 && (9.7)\\
\gamma &= \min_{i=1,2,\ldots,N} y_i\mathbf{x}_i^T\mathbf{w}\} && (9.8)
\end{aligned}
$$

và chọn $\alpha = \frac{\beta^2}{\gamma}$, ta có:

$$0 \le u_\alpha(t+1) < u_\alpha(t) + \beta^2 - 2\alpha\gamma = u_\alpha(t) - \beta^2$$

Điều này nghĩa là: nếu luôn luôn có các điểm bị misclassified thì dãy $u_\alpha(t)$ là 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 misclassified. Nói cách khác, thuật toán PLA hội tụ sau một số hữu hạn bước.

9.5 Mô hình Neural Network đầu tiên

Hàm số xác định class của Perceptron $\text{label}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T\mathbf{x})$ có thể được mô tả như hình vẽ (được gọi là network) dưới đây:

Đầu vào của network $\mathbf{x}$ được minh họa bằng các node màu xanh lục với node x_0 luôn luôn bằng 1. Tập hợp các node màu xanh lục được gọi là *Input layer*. Trong ví dụ này, tôi giả sử số chiều của dữ liệu $d = 4$. Số node trong input layer luôn luôn là $d + 1$ với một node là 1 được thêm vào. Node $x_0 = 1$ này đôi khi được ẩn đi.

Các trọng số (*weights*) $w_0, w_1, \ldots, w_d$ 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 xanh trong node y thể hiện đồ thị của hàm số sgn.

Trong thuật toán PLA, ta phải tìm các weights trên các mũi tên sao cho với mỗi $\mathbf{x}_i$ ở tập các điểm dữ liệu đã biết được đặt ở Input layer, output của network này trùng với nhãn y_i tương ứng.

Hàm số $y = \text{sgn}(z)$ còn được gọi là *activation function*. Đây chính là dạng đơn giản nhất của Neural Network.

Các Neral Networks sau này có thể có nhiều node ở output tạo thành một *output layer*, hoặc có thể có thêm các layer trung gian giữa *input layer* và *output layer*. Các layer trung gian đó được gọi là *hidden layer*. Khi biểu diễn các Networks lớn, người ta thường giản lược hình bên trái thành hình bên phải. Trong đó node $x_0 = 1$ thường được ẩn đi. Node z cũng được ẩn đi và viết gộp vào trong node y. Perceptron thường được vẽ dưới dạng đơn giản như Hình 5 bên phải.

Để ý rằng nếu ta thay *activation function* bởi $y = z$, ta sẽ có Neural Network mô tả thuật toán Linear Regression như hình dưới. 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 sau đây:

Dữ liệu từ nhiều dây thần kinh đi về một *cell nucleus*. Thông tin được tổng hợp và được đưa ra ở output. Nhiều bộ phận như thế này kết hợp với nhau tạo nên hệ thần kinh sinh học. Chính vì vậy mà có tên Neural Networks trong Machine Learning. Đôi khi mạng này còn được gọi là Artificial Neural Networks (ANN) tức *hệ neuron nhân tạo*.

9.6 Thảo Luận

9.6.1 PLA có thể cho vô số nghiệm khác nhau

Rõ ràng rằng, nếu hai class là linearly separable thì có vô số đường thẳng phân cách 2 class đó. Dưới đây là một ví dụ:

 <img src ="/assets/pla/pla6.png" align = "center" width = "400">
Hình 8: PLA có thể cho vô số nghiệm khác nhau.

Tất cả các đường thẳng màu đen đều thỏa mãn. 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? Có một thuật toán khác định nghĩa và tìm đường tốt nhất như thế, tôi sẽ giới thiệu trong 1 vài bài tới. Mời các bạn đón đọc.

9.6.2 PLA đòi hỏi dữ liệu linearly separable

Hai class trong ví dụ dưới đây *tương đối* linearly separable. Mỗi class có 1 điểm coi như *nhiễu* nằm trong khu vực các điểm của class kia. PLA sẽ không làm việc trong trường hợp này vì luôn luôn có ít nhất 2 điểm bị misclassified.

 <img src ="/assets/pla/pla7.png" align = "center" width = "400">
Hinhf 9: PLA không làm việc nếu chỉ có một nhiễu nhỏ.

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.

Để khắc phục nhược điểm này, có một cải tiến nhỏ như thuật toán Pocket Algorithm dưới đây:

9.6.3 Pocket Algorithm

Một cách tự nhiên, nếu có một vài *nhiễu*, ta sẽ đi tìm một đường thẳng phân chia hai class sao cho có ít điểm bị misclassified nhất. Việc này có thể được thực hiện thông qua PLA với một chút thay đổi nhỏ như sau:

1. Giới hạn số lượng vòng lặp của PLA. 2. Mỗi lần cập nhật nghiệm **w** mới, ta đếm xem có bao nhiêu điểm bị misclassified. Nếu là lần đầu tiên, giữ lại nghiệm này trong *pocket* (túi quần). Nếu không, so sánh số điểm misclassified này với số điểm misclassified của nghiệm trong *pocket*, nếu nhỏ hơn thì *lôi* nghiệm cũ ra, đặt nghiệm mới này vào.

Thuật toán này giống với thuật toán tìm phần tử nhỏ nhất trong 1 mảng.

9.7 Kết luận

Hy vọng rằng bài viết này sẽ giúp các bạn phần nào hiểu được một số khái niệm trong Neural Networks. Trong một số bài tiếp theo, tôi sẽ tiếp tục nói về các thuật toán cơ bản khác trong Neural Networks trước khi chuyển sang phần khác.

Trong tương lai, nếu có thể, tôi sẽ viết tiếp về Deep Learning và chúng ta sẽ lại quay lại với Neural Networks.

9.8 Tài liệu tham khảo

[1] F. Rosenblatt. The perceptron, a perceiving and recognizing automaton Project Para. Cornell Aeronautical Laboratory, 1957.

[2] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115–133, 1943.

[3] B. Widrow et al. Adaptive "Adaline" neuron using chemical "memistors". Number Technical Report 1553-2. Stanford Electron. Labs., Stanford, CA, October 1960.

[3] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. Learning from data. Vol. 4. New York, NY, USA:: AMLBook, 2012. (link to course)

[4] Bishop, Christopher M. "Pattern recognition and Machine Learning.", Springer (2006). (book)

[5] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern classification. John Wiley & Sons, 2012.

10

Logistic Regression

10.1 Giới thiệu

10.1.1 Nhắc lại hai mô hình tuyến tính

Hai mô hình tuyến tính (linear models) Linear Regression và Perceptron Learning Algorithm (PLA) chúng ta đã biết đều có chung một dạng:

$$y = f(\mathbf{w}^T\mathbf{x})$$

trong đó $f()$ được gọi là *activation function*, và $\mathbf{x}$ được hiểu là dữ liệu mở rộng với $x_0 = 1$ được thêm vào để thuận tiện cho việc tính toán. Với linear regression thì $f(s) = s$, với PLA thì $f(s) = \text{sgn}(s)$. 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 chúng ta cần dự đoán một giá trị thực của 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 *binary classification*.

Trong bài này, tôi sẽ giới thiệu mô hình thứ ba với một activation khác, được sử dụng cho các bài toán *flexible* hơn. Trong dạng này, đầu ra có thể được thể hiện dưới dạng xác suất (probability). 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,... Mô hình mới này của chúng ta có tên là *logistic regression*. Mô hình này giống với linear regression ở khía cạnh đầu ra là số thực, và giống với PLA ở việc đầu ra bị chặn (trong đoạn $[0, 1]$). Mặc dù trong tên có chứa từ *regression*, logistic regression thường được sử dụng nhiều hơn cho các bài toán classification.

10.1.2 Một ví dụ nhỏ

Tôi xin được sử dụng một ví dụ trên Wikipedia:

> Một nhóm 20 sinh viên dành thời gian trong khoảng từ 0 đến 6 giờ cho việc ôn thi. Thời gian ôn thi này ảnh hưởng đến xác suất sinh viên vượt qua kỳ thi như thế nào?

Kết quả thu được như sau:

| Hours | Pass | Hours | Pass | | :——-: | :——: | :——-: | :——: | | .5 | 0 | 2.75 | 1 | | .75 | 0 | 3 | 0 | | 1 | 0 | 3.25 | 1 | | 1.25 | 0 | 3.5 | 0 | | 1.5 | 0 | 4 | 1 | | 1.75 | 0 | 4.25 | 1 | | 1.75 | 1 | 4.5 | 1 | | 2 | 0 | 4.75 | 1 | | 2.25 | 1 | 5 | 1 | | 2.5 | 0 | 5.5 | 1 |

Mặc dù có một chút *bất công* khi học 3.5 giờ thì trượt, còn học 1.75 giờ thì lại đỗ, nhìn chung, học càng nhiều thì khả năng đỗ càng cao. PLA không thể áp dụng được cho bài toán này vì không thể nói một người học bao nhiêu giờ thì 100

Chúng ta biểu diễn các điểm này trên đồ thị để thấy rõ hơn:

Nhận thấy rằng cả linear regression và PLA đều không phù hợp với bài toán này, chúng ta cần một mô hình *flexible* hơn.

10.1.3 Mô hình Logistic Regression

Đầu ra dự đoán của:

- Linear Regression:
$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$
- PLA:
$$f(\mathbf{x}) = \text{sgn}(\mathbf{w}^T\mathbf{x})$$

Đầu ra dự đoán của logistic regression thường được viết chung dưới dạng:

$$f(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x})$$

Trong đó θ được gọi là logistic function. Một số activation cho mô hình tuyến tính được cho trong hình dưới đây:

- Đường màu vàng biểu diễn linear regression. Đường này không bị chặn nên không phù hợp cho bài toán này. Có một *trick* nhỏ để đưa nó về dạng bị chặn: *cắt* phần nhỏ hơn 0 bằng cách cho chúng bằng 0, *cắt* các phần lớn hơn 1 bằng cách cho chúng bằng 1. Sau đó lấy điểm trên đường thẳng này có tung độ bằng 0.5 làm điểm phân chia hai *class*, đây cũng không phải là một lựa chọn tốt. Giả sử có thêm vài bạn *sinh viên tiêu biểu* ôn tập đến 20 giờ và, tất nhiên, thi đỗ. Khi áp dụng mô hình linear regression như hình dưới đây và lấy mốc 0.5 để phân lớp, toàn bộ sinh viên thi trượt vẫn được dự đoán là trượt, nhưng rất nhiều sinh viên thi đỗ cũng được dự đoán là trượt (nếu ta coi điểm x màu xanh lục là *ngưỡng cứng* để đưa ra kết luận). Rõ ràng đây là một mô hình không tốt. Anh chàng sinh viên tiêu biểu này đã *kéo theo* rất nhiều bạn khác bị trượt.

- Đường màu đỏ (chỉ khác với activation function của PLA ở chỗ hai class là 0 và 1 thay vì -1 và 1) cũng thuộc dạng *ngưỡng cứng* (hard threshold). PLA không hoạt động trong bài toán này vì dữ liệu đã cho không *linearly separable*.

- Các đường màu xanh lam và xanh lục phù hợp với bài toán của chúng ta hơn. Chúng có một vài tính chất quan trọng sau:

  – Là hàm số liên tục nhận giá trị thực, bị chặn trong khoảng $(0, 1)$.

  – Nếu coi điểm có tung độ là $1/2$ làm điểm phân chia thì các điểm càng xa điểm này về phía bên trái có giá trị càng gần 0. Ngược lại, các điểm càng xa điểm này về phía 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.

  – *Mượt* (smooth) nên có đạo hàm mọi nơi, có thể được lợi trong việc tối ưu.

10.1.4 Sigmoid function

Trong số các hàm số có 3 tính chất nói trên thì hàm *sigmoid*:

$$f(s) = \frac{1}{1 + e^{-s}} \triangleq \sigma(s)$$

được sử dụng nhiều nhất, vì nó bị chặn trong khoảng $(0, 1)$. Thêm nữa:

$$\lim_{s \to -\infty} \sigma(s) = 0; \quad \lim_{s \to +\infty} \sigma(s) = 1$$

Đặc biệt hơn nữa:

$$\begin{aligned} \sigma'(s) &= \frac{e^{-1}}{(1 + e^{-s})^2} \\ &= \frac{1}{1 + e^{-s}} \frac{e^{-s}}{1 + e^{-s}} \\ &= \sigma(s)(1 - \sigma(s)) \end{aligned}$$

Công thức đạo hàm đơn giản thế này giúp hàm số này được sử dụng rộng rãi. Ở phần sau, tôi sẽ lý giải việc *người ta đã tìm ra hàm số đặc biệt này 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 $(-1, 1)$ nhưng có thể dễ dàng đưa nó về khoảng $(0, 1)$. Bạn đọc có thể chứng minh được:

$$\tanh(s) = 2\sigma(2s) - 1$$

10.2 Hàm mất mát và phương pháp tối ưu

10.2.1 Xây dựng hàm mất mát

Với mô hình như trên (các activation màu xanh lam và lục), ta có thể giả sử rằng xác suất để một điểm dữ liệu $\mathbf{x}$ rơi vào class 1 là $f(\mathbf{w}^T\mathbf{x})$ và rơi vào class 0 là $1 - f(\mathbf{w}^T\mathbf{x})$. Với mô hình được giả sử như vậy, với các điểm dữ liệu training (đã biết đầu ra y), ta có thể viết như sau:

$$P(y_i = 1|\mathbf{x}_i; \mathbf{w}) = f(\mathbf{w}^T\mathbf{x}_i)$$
$$P(y_i = 0|\mathbf{x}_i; \mathbf{w}) = 1 - f(\mathbf{w}^T\mathbf{x}_i)$$

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 $\mathbf{x}_i$. Bạn đọc có thể đọc thêm Xác suất có điều kiện. Mục đích của chúng ta là tìm các hệ số $\mathbf{w}$ sao cho $f(\mathbf{w}^T\mathbf{x}_i)$ càng gần với 1 càng tốt với các điểm dữ liệu thuộc class 1 và càng gần với 0 càng tốt với những điểm thuộc class 0.

Ký hiệu $z_i = f(\mathbf{w}^T\mathbf{x}_i)$ và viết gộp lại hai biểu thức bên trên ta có:

$$P(y_i|\mathbf{x}_i; \mathbf{w}) = z_i^{y_i}(1 - z_i)^{1-y_i}$$

Biểu thức này là tương đương với hai biểu thức (1) và (2) ở trên vì khi $y_i = 1$, phần thứ hai của vế phải sẽ triệt tiêu, khi $y_i = 0$, phần thứ nhất sẽ bị triệt tiêu! Chúng ta muốn mô hình gần với dữ liệu đã cho nhất, tức xác suất này đạt giá trị cao nhất.

Xét toàn bộ training set với $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N] \in \mathbb{R}^{d\times N}$ và $\mathbf{y} = [y_1, y_2, \ldots, y_N]$, chúng ta cần tìm $\mathbf{w}$ để biểu thức sau đây đạt giá trị lớn nhất:

$$P(\mathbf{y}|\mathbf{X}; \mathbf{w})$$

ở đây, ta cũng ký hiệu $\mathbf{X}, \mathbf{y}$ như các biến ngẫu nhiên (random variables). Nói cách khác:

$$\mathbf{w} = \arg\max_{\mathbf{w}} P(\mathbf{y}|\mathbf{X}; \mathbf{w})$$

Bài toán tìm tham số để mô hình gần với dữ liệu nhất trên đây có tên gọi chung là bài toán *maximum likelihood estimation* với hàm số phía sau arg max được gọi là *likelihood function*. Khi làm việc với các bài toán Machine Learning sử dụng các mô hình xác suất thống kê, chúng ta sẽ gặp lại các bài toán thuộc dạng này, hoặc *maximum a posteriori estimation*, rất nhiều. Tôi sẽ dành 1 bài khác để nói về hai dạng bài toán này.

Giả sử thêm rằng các điểm dữ liệu được sinh ra một cách ngẫu nhiên độc lập với nhau (independent), ta có thể viết:

$$P(\mathbf{y}|\mathbf{X};\mathbf{w}) = \prod_{i=1}^{N} P(y_i|\mathbf{x}_i;\mathbf{w}) = \prod_{i=1}^{N} z_i^{y_i}(1-z_i)^{1-y_i} \tag{10.1}$$

với $\prod$ là ký hiệu của tích. Bạn đọc có thể muốn đọc thêm về Độc lập thống kê.

Trực tiếp tối ưu hàm số này theo $\mathbf{w}$ nhìn qua không đơn giản! Hơn nữa, khi N lớn, tích của N số nhỏ hơn 1 có thể dẫn tới sai số trong tính toán (numerial error) vì tích là một số quá nhỏ. Một phương pháp thường được sử dụng đó là lấy logarit tự nhiên (cơ số e) của *likelihood function* biến phép nhân thành phép cộng và để tránh việc số quá nhỏ. Sau đó lấy ngược dấu để được một hàm và coi nó là hàm mất mát. Lúc này bài toán tìm giá trị lớn nhất (maximum likelihood) trở thành bài toán tìm giá trị nhỏ nhất của hàm mất mát (hàm này còn được gọi là negative log likelihood):

$$J(\mathbf{w}) = -\log P(\mathbf{y}|\mathbf{X};\mathbf{w}) = -\sum_{i=1}^{N}(y_i \log z_i + (1-y_i)\log(1-z_i)) \tag{10.2}$$

với chú ý rằng z_i là một hàm số của $\mathbf{w}$. Bạn đọc tạm nhớ biểu thức vế phải có tên gọi là *cross entropy*, thường được sử dụng để đo *khoảng cách* giữa hai phân phối (distributions). Trong bài toán đang xét, một phân phối là dữ liệu được cho, với xác suất chỉ là 0 hoặc 1; phân phối còn lại được tính theo mô hình logistic regression. *Khoảng cách* giữa hai phân phối nhỏ đồng nghĩa với việc (*có vẻ hiển nhiên là*) hai phân phối đó rất gần nhau. Tính chất cụ thể của hàm số này sẽ được đề cập trong một bài khác mà tầm quan trọng của khoảng cách giữa hai phân phối là lớn hơn.

Chú ý: Trong machine learning, logarit thập phân ít được dùng, vì vậy log thường được dùng để ký hiệu logarit tự nhiên.

10.2.2 Tối ưu hàm mất mát

–>

Chúng ta lại sử dụng phương pháp Stochastic Gradient Descent (SGD) ở đây (*Bạn đọc được khuyến khích đọc SGD trước khi đọc phần này*) . Hàm mất mát với chỉ một điểm dữ liệu $(\mathbf{x}_i, y_i)$ là:

$$J(\mathbf{w};\mathbf{x}_i,y_i) = -(y_i \log z_i + (1-y_i)\log(1-z_i))$$

Với đạo hàm:

$$\frac{\partial J(\mathbf{w};\mathbf{x}_i,y_i)}{\partial \mathbf{w}} = -(\frac{y_i}{z_i} - \frac{1-y_i}{1-z_i})\frac{\partial z_i}{\partial \mathbf{w}} = \frac{z_i - y_i}{z_i(1-z_i)}\frac{\partial z_i}{\partial \mathbf{w}} \qquad (3) \tag{10.3}$$

Để cho biểu thức này trở nên *gọn* và *đẹp* hơn, chúng ta sẽ tìm hàm $z = f(\mathbf{w}^T\mathbf{x})$ sao cho mẫu số bị triệt tiêu. Nếu đặt $s = \mathbf{w}^T\mathbf{x}$, chúng ta sẽ có:

$$\frac{\partial z_i}{\partial \mathbf{w}} = \frac{\partial z_i}{\partial s}\frac{\partial s}{\partial \mathbf{w}} = \frac{\partial z_i}{\partial s}\mathbf{x}$$

Một cách trực quan nhất, ta sẽ tìm hàm số $z = f(s)$ sao cho:

$$\frac{\partial z}{\partial s} = z(1-z) \quad (4)$$

để triệt tiêu mẫu số trong biểu thức (3). Chúng ta cùng khởi động một chút với phương trình vi phân đơn giản này. Phương trình (4) tương đương với:

$$\begin{aligned}
& \tfrac{\partial z}{z(1-z)} = && \partial s \\
\Leftrightarrow \quad & (\tfrac{1}{z} + \tfrac{1}{1-z})\partial z = && \partial s \\
\Leftrightarrow & \log z - \log(1-z) = s \\
\Leftrightarrow & \log \tfrac{z}{1-z} = && s \\
\Leftrightarrow & \tfrac{z}{1-z} = && e^s \\
\Leftrightarrow & z = && e^s(1-z) \\
\Leftrightarrow & z = \tfrac{e^s}{1+e^s} = && \frac{1}{1+e^{-s}} = \sigma(s)
\end{aligned}$$

Đến đây, tôi hy vọng các bạn đã hiểu hàm số *sigmoid* được tạo ra như thế nào.

Chú ý: Trong việc giải phương trình vi phân ở trên, tôi đã bỏ qua hằng số khi lấy nguyên hàm hai vế. Tuy vậy, việc này không ảnh hưởng nhiều tới kết quả.

10.2.3 Công thức cập nhật cho logistic sigmoid regression

Tới đây, bạn đọc có thể kiểm tra rằng:

$$\frac{\partial J(\mathbf{w}; \mathbf{x}_i, y_i)}{\partial \mathbf{w}} = (z_i - y_i)\mathbf{x}_i$$

Qúa đẹp!

Và công thức cập nhật (theo thuật toán SGD) cho logistic regression là:

$$\mathbf{w} = \mathbf{w} + \eta(y_i - z_i)\mathbf{x}_i$$

Khá đơn giản! Và, như thường lệ, chúng ta sẽ có vài ví dụ với Python.

10.3 Ví dụ với Python

10.3.1 Ví dụ với dữ liệu 1 chiều

Quay trở lại với ví dụ nêu ở phần Giới thiệu. Trước tiên ta cần khai báo vài thư viện và dữ liệu:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(2)

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]])
y = np.array([0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1])

# extened data
X = np.concatenate((np.ones((1, X.shape[1])), X), axis = 0)
```

10.3.2 Các hàm cần thiết cho logistic sigmoid regression

```
def sigmoid(s):
    return 1/(1 + np.exp(-s))

def logistic_sigmoid_regression(X, y, w_init, eta, tol = 1e-4, max_count = 10000):
    w = [w_init]
    it = 0
    N = X.shape[1]
    d = X.shape[0]
    count = 0
    check_w_after = 20
    while count < max_count:
        # mix data
        mix_id = np.random.permutation(N)
        for i in mix_id:
            xi = X[:, i].reshape(d, 1)
            yi = y[i]
            zi = sigmoid(np.dot(w[-1].T, xi))
            w_new = w[-1] + eta*(yi - zi)*xi
            count += 1
            # stopping criteria
            if count%check_w_after == 0:
                if np.linalg.norm(w_new - w[-check_w_after]) < tol:
                    return w
            w.append(w_new)
    return w
eta = .05
d = X.shape[0]
w_init = np.random.randn(d, 1)

w = logistic_sigmoid_regression(X, y, w_init, eta)
print(w[-1])
```

[[-4.092695] [1.55277242]]

Với kết quả tìm được, đầu ra y có thể được dự đoán theo công thức: `y = sigmoid(−4.1 + 1.55*x)`. Với dữ liệu trong tập training, kết quả là:

```
print(sigmoid(np.dot(w[-1].T, X)))
```

```
    [[ 0.03281144  0.04694533  0.06674738  0.09407764  0.13102736  0.17961209
       0.17961209  0.24121129  0.31580406  0.40126557  0.49318368  0.58556493
       0.67229611  0.74866712  0.86263755  0.90117058  0.92977426  0.95055357
       0.96541314  0.98329067]]
```

Biểu diễn kết quả này trên đồ thị ta có:

```
X0 = X[1, np.where(y == 0)][0]
y0 = y[np.where(y == 0)]
X1 = X[1, np.where(y == 1)][0]
y1 = y[np.where(y == 1)]

plt.plot(X0, y0, 'ro', markersize = 8)
plt.plot(X1, y1, 'bs', markersize = 8)

xx = np.linspace(0, 6, 1000)
w0 = w[-1][0][0]
w1 = w[-1][1][0]
threshold = -w0/w1
yy = sigmoid(w0 + w1*xx)
plt.axis([-2, 8, -1, 2])
plt.plot(xx, yy, 'g-', linewidth = 2)
plt.plot(threshold, .5, 'y^', markersize = 8)
plt.xlabel('studying hours')
plt.ylabel('predicted probability of pass')
plt.show()
```

Nếu như chỉ có hai output là 'fail' hoặc 'pass', điểm trên đồ thị của hàm sigmoid tương ứng với xác suất 0.5 được chọn làm *hard threshold* (ngưỡng cứng). Việc này có thể chứng minh khá dễ dàng (tôi sẽ bàn ở phần dưới).

10.3.3 Ví dụ với dữ liệu 2 chiều

Chúng ta xét thêm một ví dụ nhỏ nữa trong không gian hai chiều. Giả sử chúng ta có hai class xanh-đỏ với dữ liệu được phân bố như hình dưới. Với dữ liệu đầu vào nằm trong không gian hai chiều, hàm sigmoid có dạng như thác nước dưới đây:

Kết quả tìm được khi áp dụng mô hình logistic regression được minh họa như hình dưới với màu nền khác nhau thể hiện xác suất điểm đó thuộc class đỏ. Đỏ hơn tức gần 1 hơn, xanh hơn tức gần 0 hơn.

Nếu phải lựa chọn một *ngưỡng cứng* (chứ không chấp nhận xác suất) để phân chia hai class, chúng ta quan sát thấy đường thẳng nằm nằm trong khu vực xanh lục là một lựa chọn hợp lý. Tôi sẽ chứng minh ở phần dưới rằng, đường phân chia giữa hai class tìm được bởi logistic regression có dạng một đường phẳng, tức vẫn là linear.

10.4 Một vài tính chất của Logistic Regression

10.4.1 Logistic Regression thực ra được sử dụng nhiều trong các bài toán Classification.

Mặc dù có tên là Regression, tức một mô hình cho fitting, Logistic Regression lại 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:

$$P(y = 1|\mathbf{x}; \mathbf{w}); \quad P(y = 0|\mathbf{x}; \mathbf{w})$$

Nếu biểu thức thứ nhất lớn hơn thì ta kết luận điểm dữ liệu thuộc class 1, ngược lại thì nó thuộc class 0. Vì tổng hai biểu thức này luôn bằng 1 nên một cách gọn hơn, ta chỉ cần xác định xem $P(y = 1|\mathbf{x}; \mathbf{w})$ lớn hơn 0.5 hay không. Nếu có, class 1. Nếu không, class 0.

10.4.2 Boundary tạo bởi Logistic Regression có dạng tuyến tính

Thật vậy, theo lập luận ở phần trên thì chúng ta cần kiểm tra:

$$\begin{aligned} & P(y = 1|\mathbf{x}; \mathbf{w}) > 0.5 \\ \Leftrightarrow & \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} > 0.5 \\ \Leftrightarrow & e^{-\mathbf{w}^T\mathbf{x}} < 1 \\ \Leftrightarrow & \mathbf{w}^T\mathbf{x} > 0 \end{aligned}$$

Nói cách khác, boundary giữa hai class là đường có phương trình $\mathbf{w}^T\mathbf{x}$. Đây chính là phương trình của một siêu mặt phẳng. Vậy Logistic Regression tạo ra boundary có dạng tuyến tính.

10.5 Thảo luận

- Một điểm cộng cho Logistic Regression so với PLA là nó không cần có giả thiết dữ liệu hai class là linearly separable. Tuy nhiên, boundary tìm được vẫn có dạng tuyến tính. Vậy nên mô hình này chỉ phù hợp với loại dữ liệu mà hai class là gần với linearly separable. Một kiểu dữ liệu mà Logistic Regression không làm việc được là dữ liệu mà một class chứa các điểm nằm trong 1 vòng tròn, class kia chứa các điểm bên ngoài đường tròn đó. Kiểu dữ liệu này được gọi là phi tuyến (non-linear). Sau một vài bài nữa, tôi sẽ giới thiệu với các bạn các mô hình khác phù hợp hơn với loại dữ liệu này hơn.

- Một hạn chế nữa của Logistic Regression là nó yêu cầu các điểm dữ liệu được tạo ra một cách *độc lập* với nhau. Trên thực tế, các điểm dữ liệu có thể bị *ảnh hưởng* bởi nhau. Ví dụ: có một nhóm ôn tập với nhau trong 4 giờ, cả nhóm đều thi đỗ (giả sử các bạn này học rất tập trung), nhưng có một sinh viên học một mình cũng trong 4 giờ thì xác suất thi đỗ thấp hơn. Mặc dù vậy, để cho đơn giản, khi xây dựng mô hình, người ta vẫn thường giả sử các điểm dữ liệu là độc lập với nhau.

- Khi biểu diễn theo Neural Networks, Linear Regression, PLA, và Logistic Regression có dạng như sau:

- Nếu hàm mất mát của Logistic Regression được viết dưới dạng:

$$J(\mathbf{w}) = \sum_{i=1}^{N} (y_i - z_i)^2$$

  thì khó khăn gì sẽ xảy ra? Các bạn hãy coi đây như một bài tập nhỏ.

- Source code cho các ví dụ trong bài này có thể tìm thấy ở đây.

10.6 Tài liệu tham khảo

[1] Cox, David R. "The regression analysis of binary sequences." Journal of the Royal Statistical Society. Series B (Methodological) (1958): 215-242.

[2] Cramer, Jan Salomon. "The origins of logistic regression." (2002).

[3] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, and Hsuan-Tien Lin. Learning from data. Vol. 4. New York, NY, USA:: AMLBook, 2012. (link to course)

[4] Bishop, Christopher M. "Pattern recognition and Machine Learning.", Springer (2006). (book)

[5] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern classification. John Wiley & Sons, 2012.

[6] Andrer Ng. CS229 Lecture notes. Part II: Classification and logistic regression

[7] Jerome H. Friedman, Robert Tibshirani, and Trevor Hastie. The Elements of Statistical Learning.

11

Giới thiệu về Feature Engineering

11.1 Giới thiệu

Cho tới lúc này, tôi đã trình bày 5 thuật toán Machine Learning cơ bản: Linear Regression, K-means Clusterning, K-nearest neighbors, Perceptron Learning Algorithm và Logistic Regression. Trong tất cả các thuật toán này, tôi đều giả sử các điểm dữ liệu được biểu diễn bằng các vector, được gọi là *feature vector* hay *vector đặc trưng*, có độ dài bằng nhau, và cùng là vector cột hoặc vector hàng. Tuy nhiên, trong các bài toán thực tế, mọi chuyện không được tốt đẹp như vậy!

Với các bài toán về Computer Vision, các bức ảnh là các ma trận có kích thước khác nhau. Thậm chí để nhận dạng vật thể trong ảnh, ta cần thêm một bước nữa là *object detection*, tức là tìm cái khung chứa vật thể chúng ta cần dự đoán. Ví dụ, trong bài toán nhận dạng khuôn mặt, chúng ta cần tìm được vị trí các khuôn mặt trong ảnh và *crop* các khuôn mặt đó trước khi làm các bước tiếp theo. Ngay cả khi đã xác định được các khung chứa các khuôn mặt (và có thể resize các khung đó về cùng một kích thước), ta vẫn phải làm rất nhiều việc nữa vì hình ảnh của khuôn mặt còn phụ thuộc vào góc chụp, ánh sáng, ... và rất nhiều yếu tố khác nữa.

Các bài toán NLP (Natural Language Processing - Xử lý ngôn ngữ tự nhiên) 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, thậm chí của cùng một người nhưng lúc ốm lúc khỏe cũng khác nhau.

Khi làm việc với các bài toán Machine Learning thực tế, nhìn chung chúng ta chỉ có được dữ liệu thô (raw) chưa qua chỉnh sửa, chọn lọc. Chúng ta cần phải tìm một phép biến đổi để loại ra những dữ liệu nhiễu (noise), và để đưa dữ liệu thô với 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 (features) 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 features phù hợp. Quá trình

quan trọng này được gọi là *Feature Extraction*, hoặc *Feature Engineering*, một số tài liệu tiếng Việt gọi nó là *trích chọn đặc trưng*.

Tôi xin trích một câu nói của thầy Andrew Ng và xin phép thêm không dịch ra tiếng Việt (Nguồn Feature Engineering - wiki):

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

Để giúp các bạn có cái nhìn tổng quan hơn, trong phần tiếp theo tôi xin đặt bước Feature Engineering này trong một bức tranh lớn hơn.

11.2 Mô hình chung cho các bài toán Machine Learning

Phần lớn các bài toán Machine Learning có thể được thể hiện trong hình (tôi) vẽ dưới đây:

Có hai phases lớn là Training phase và Testing phase. Xin nhắc lại là với các bài toán Supervised learning, ta có các cặp dữ liệu (*input, output*), với các bài toán Unsupervised learing, ta chỉ có *input* mà thôi.

11.2.1 TRAINING PHASE

Có hai khối có nền màu xanh lục chúng ta cần phải thiết kế:

Feature Extractor

ĐẦU RA

Tôi xin đề cập đầu ra của khối này trước vì mục đích của Feature Engineering là tạo ra một Feature Extractor biến dữ liệu thô ban đầu thành dữ liệu phù hợp với từng mục đích khác nhau.

ĐẦU VÀO

- ***raw training input***. Raw input là tất cả các thông tin ta biết về dữ liệu. Ví dụ: với ảnh thì là giá trị của từng pixel; với văn bản thì là từng từ, từng câu; với file âm thanh thì nó là một đoạn tín hiệu; với cơ sở dữ liệu Iris thì nó là độ dài các cánh hoa và đài hoa, ... Dữ liệu thô này thường không ở dạng vector, không có số chiều như nhau. Thậm chí có thể có số chiều như nhau nhưng số chiều quá lớn, như một bức ảnh màu 1000 pixel x 1000 pixel thì số *elements* đã là 3×10^6 (3 vì ảnh màu thường có 3 channels: Red, Green, Blue). Đây là một con số quá lớn, không lợi cho lưu trữ và tính toán.

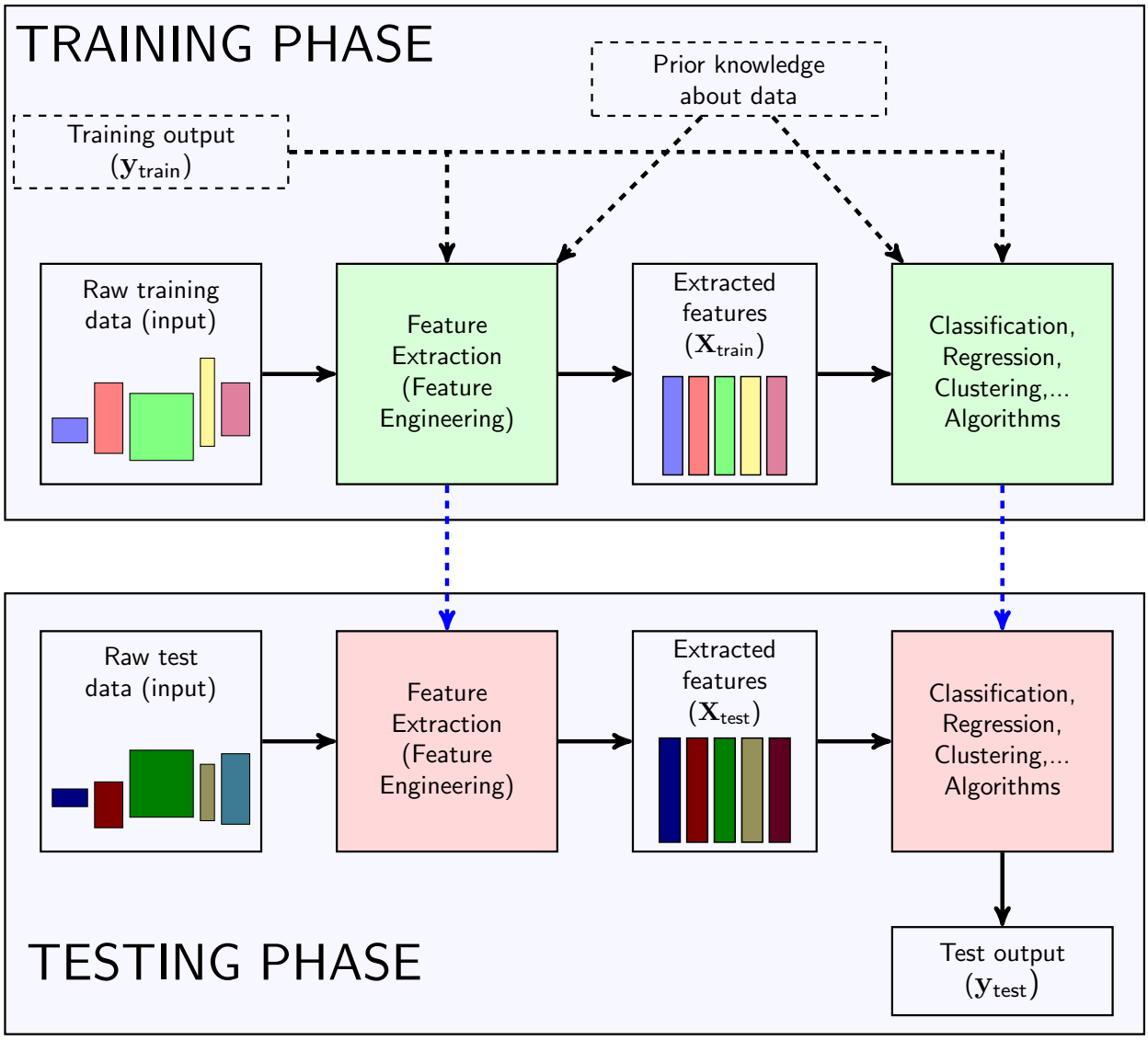

Fig. 11.1: Mô hình chung cho các bài toán Machine Learning.

- **(optional) *output* của *training set***. Trong các bài toán Unsupervised learning, ta không biết *output* nên hiển nhiên sẽ không có đầu vào 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. Ma trận này thường là ma trận *béo* (số hàng ít hơn số cột, tiếng Anh - fat matrices) 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 *raw input*, ta sẽ có bài toán tìm ma trận chiếu để lượng thông tin mất đi là ít nhất. Trong nhiều trường hợp, dữ liệu *output* của *training set* cũng được sử dụng để tạo ra Feature Extractor. Ví dụ: trong bài toán classification,

ta không quan tâm nhiều đến việc mất thông tin hay không, ta chỉ quan tâm đến việc những thông tin còn lại có đặc trưng cho từng class hay không. Ví dụ, dữ liệu thô là các hình vuông và hình tam giác có màu đỏ và xanh. Trong bài toán phân loại đa giác, các output là *tam giác* và *vuông*, thì 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 class là *xanh* và *đỏ*, ta không quan tâm tới số cạnh mà chỉ quan tâm đến màu sắc thôi.

- **(optional) *Prior knowledge about data***: Dôi khi những giả thiết khác về dữ liệu cũng mang lại lợi ích. Ví dụ, trong bài toán classification, nếu ta biết dữ liệu là (gần như) *linearly separable* thì ta sẽ đi tìm một ma trận chiếu sao cho ở trong không gian mới, dữ liệu vẫn đảm bảo tính *linearly separable*, việc này thuận tiện hơn cho phần classification vì các thuật toán linear, nhìn chung, đơn giản hơn.

Sau khi *học* được feature extractor thì ta cũng sẽ thu được *extracted features* cho *raw input data*. Những *extracted features* này được dùng để huấn luyện các thuật toán Classification, Clustering, Regression,... ở phía sau.

Main Algorithms

Khi có được *extracted features* rồi, chúng ta sử dụng những thông tin này cùng với (optional) *training output* và (optional) *prior knowledge* để tạo ra các mô hình phù hợp, điều mà chúng ta đã làm ở những bài trước.

Chú ý: Trong một số thuật toán cao cấp hơn, việc *huấn luyện* feature extractor và main algorithm được thực hiện cùng lúc với nhau chứ không phải từng bước như trên.

Một điểm rất quan trọng: khi xây dựng bộ *feature extractor* và *main algorithms*, chúng ta không được sử dụng bất kỳ thông tin nào trong tập *test data*. Ta phải giả sử rằng những thông tin trong *test data* chưa được nhìn thấy bao giờ. Nếu sử dụng thêm thông tin về *test data* thì rõ ràng ta đã *ăn gian*! Tôi từng đánh giá các bài báo khoa học quốc tế, rất nhiều tác giả xây dựng mô hình dùng cả dữ liệu *test data*, sau đó lại dùng chính mô hình đó để kiểm tra trên *test data* đó. Việc *ăn gian* này là lỗi rất nặng và hiển nhiên những bài báo đó bị từ chối (reject).

11.2.2 TESTING PHASE

Bước này đơn giản hơn nhiều. Với *raw input* mới, ta sử dụng feature extractor đã tạo được ở trên (tất nhiên không được sử dụng *output* của nó vì *output* là cái ta đang đi tìm) để tạo ra feature vector tương ứng. Feature vector được đưa vào *main algorithm* đã được học ở training phase để dự đoán *output*.

11.3 Một số ví dụ về Feature Engineering

11.3.1 Trực tiếp lấy raw data

Với bài toán phân loại chữ số viết tay trong bộ cơ sở dữ liệu MNIST, mỗi bức ảnh có số chiều là 28 pixel x 28 pixel (tất nhiên việc *crop* và chỉnh sửa mỗi bức ảnh đã được thực hiện từ trước rồi, đó đã là một phần của feature engineering rồi). Một cách đơn giản thường được dùng là *kéo dài* ma trận 28x28 này để được 1 vector có số chiều 784. Trong cách này, các cột (hoặc hàng) của ma trận ảnh được đặt chồng lên (hoặc cạnh nhau) để được 1 vector dài. Vector dài này được trực tiếp sử dụng làm feature đưa vào các bộ classifier/clustering/regression/... Lúc này, giá trị của mỗi pixel ảnh được coi là một feature.

Rõ ràng việc làm đơn giản này đã làm mất thông tin về *không gian* (spatial information) giữa các điểm ảnh, tuy nhiên, trong nhiều trường hợp, nó vẫn mang lại kết quả khả quan.

11.3.2 Feature selection

Giả sử rằng các điểm dữ liệu có số features khác nhau (do kích thước dữ liệu khác nhau hay do một số feature mà điểm dữ liệu này có nhưng điểm dữ liệu kia lại không thu thập được), và số lượng features là cực lớn. Chúng ta cần *chọn* ra một số lượng nhỏ hơn các feature phù hợp với bài toán. *Chọn thế nào* và *thế nào là phù hợp* lại là một bài toán khác, tôi sẽ không bàn thêm ở đây.

11.3.3 Dimensionality reduction

Một phương pháp nữa tôi đã đề cập đó là làm giảm số chiều của dữ liệu để giảm bộ nhớ và khối lượng tính toán. Việc giảm số chiều này có thể được thực hiện bằng nhiều cách, trong đó *random projection* là cách đơn giản nhất. Tức chọn một *ma trận chiếu* (projection matrix) ngẫu nhiên (ma trận béo) rồi nhân nó với từng điểm dữ liệu (giả sử dữ liệu ở dạng vector cột) để được các vector có số chiều thấp hơn. Lúc này, có thể ta không có tên gọi cho mỗi feature nữa vì các feature ở vector ban đầu đã được trộn lẫn với nhau theo một tỉ lệ nào đó rồi lưu và vector mới này. Mỗi thành phần của vector mới này được coi là một feature (không tên).

Việc chọn một ma trận chiếu ngẫu nhiên đôi khi mang lại kết quả tệ không mong muốn vì thông tin bị mất đi quá nhiều. Một phương pháp được sử dụng nhiều để hạn chế lượng thông tin mất đi có tên là Principle Component Analysis sẽ được tôi trình bày sau đây khoảng 1-2 tháng.

Chú ý: Feature learning không nhất thiết phải làm giảm số chiều dữ liệu, đôi khi feature vector còn có số chiều lớn hơn raw data. Random projection cũng có thể làm được việc này nếu ma trận chiếu là một ma trận *cao* (số cột ít hơn số hàng).

11.3.4 Bag-of-words

Hẳn rất nhiều bạn đã tự đặt câu hỏi: Với một văn bản thì feature vector 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 phương pháp rất phổ biến giúp bạn trả lời những câu hỏi này. Phương pháp đó có tên là *Bag of Words (BoW)* (*Túi đựng Từ*).

Vẫn theo thói quen, tôi bắt đầu bằng một ví dụ. Giả sử chúng ta có bài toán phân loại tin rác. Ta thấy rằng nếu một tin có chứa các từ *khuyến mại, giảm giá, trúng thưởng, miễn phí, quà tặng, tri ân, ...* thì nhiều khả năng đó là một tin nhắn rác. Vậy phương pháp đơn giản nhất là *đếm* xem trong tin đó có bao nhiêu từ thuộc vào các từ trên, nếu nhiều hơn 1 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 thì 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 làm các vector đặc trưng cho từng văn bản.

Tôi xin lấy ví dụ cụ thể hơn về cách tạo ra vector đặc trưng cho mỗi văn bản dựa trên BoW và xin được lấy tiếng Anh làm ví dụ (nguồn Bag of Words wiki. Tiếng Việt khó hơn vì một từ có thể có nhều âm tiết, tiếng Anh thì thường cứ gặp dấu cách là kết thúc một từ).

Giả sử chúng ta có hai văn bản đơn giản:

```
(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* với 10 *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 là:

```
(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", ... 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 10 từ rất nhiều, có thể đến một trăm nghìn hoặc cả triệu, như vậy vector đặc trưng thu được sẽ rất *dài*. Một văn bản chỉ có 1 câu, và 1 tiểu thuyết nghìn trang đều được biểu diễn bằng các vector có số chiều bằng 100 nghìn hoặc 1 triệu.

- 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 0. Các vector có nhiều phần tử bằng 0 được gọi là *sparse vector* (sparse hiểu theo nghĩa là *thưa thớt, rải rác*, tôi xin phép chỉ sử dụng khái niệm này bằng tiếng Anh). 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. Lưu ý: nếu có hơn 50

- Thi thoảng có những từ hiếm gặp không nằm trong từ điển, vậy ta sẽ làm gì? Một cách thường được dùng là *mở rộng* vector đặc trưng thêm 1 phần tử, gọi là phần tử `<Unknown>`. Mọi từ không có trong từ điển đều được coi là `<Unknown>`.

- Nghĩ kỹ một chút, những từ hiếm đôi khi lại mang những thông tin qua trọng nhất mà chỉ loại văn bản đó có. Đây là một nhược điểm của BoW. Có một phương pháp cải tiến khác giúp khắc phục nhược điểm này có tên là Term Frequency-Inverse Document Frequency (TF-IDF) dùng để xác định tầm quan trọng của một từ trong một văn bản dựa trên toàn bộ văn bản trong cơ sở dữ liệu (corpus). Bạn đọc muốn tìm hiểu thêm có thể xem 5 Algorithms Every Web Developer Can Use and Understand, section 5.

- 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. 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.

Bonus: hình dưới đay là tần suất sử dụng các từ (coi mỗi âm tiết là một từ) trong Truyện Kiều (theo bản này) nếu ta chỉ sử dụng 30 từ có tần suất cao nhất. :

11.3.5 Bag-of-Words trong Computer Vision

Bags of Words cũng được áp dụng trong Computer Vision với cách định nghĩa *words* 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. Phân loại một bức ảnh là rừng hay sa mạc (giả sử ta biết rằng nó thuộc một trong hai loại này) một cách trực quan nhất là dựa vào màu sắc. Màu xanh 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 3 màu xanh (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 1; gần đỏ nhất, tăng x_2 lên 1; gần vàng nhất, tăng x_3 lên 1.

- 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 độ dài bằng 3, mỗi phần tử thể hiện việc có bao nhiêu pixel trong bức ảnh có màu tương ứng. Vector cuối này còn được gọi là vector histogram của bức ảnh tương ứng với ba màu xanh, đỏ, vàng. Dựa vào vector này, ta có thể quyết định bức ảnh đó là ảnh rừng hay sa 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. Một cái (ảnh) cây dù không có màu vẫn là một cái (ảnh) cây! Vì vậy, xem xét giá trị từng điểm ảnh một không mang lại kết quả khả quan vì lượng thông tin bị mất quá nhiều.

Có một cách khắc phục là thay vì xem xét một điểm ảnh, ta xem xét một *cửa sổ* nhỏ trong ảnh (trong Computer Vision, cửa sổ này được gọi là patch) là một hình chữ nhật chứa nhiều điểm ảnh gần nhau. Cửa sổ này 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 hình dưới đây.

Tương tự thế, với ảnh là ô tô, các patch thu được có thể là bánh xe, khung xe, cửa xe, ... như hàng trên trong hình dưới đây.

Có một câu hỏi 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 nau. Vậy trong xử lý ảnh, hai patchés được coi là như nhau khi nào? Khi mọi pixel trong hai patches có giá trị bằng nhau sao?

Câu trả lời là không. Xác suất để hai patches 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 ta vẫn nhìn thấy hai patches 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: hai patches là gần giống nhau nếu khoảng cách Euclid giữa hai vector tạo bởi hai patches đó gần nhau. Từ điển (codebook) sẽ có số phần tử do ta tự chọn. Số phần tử càng cao thì độ sai lệch càng ít, nhưng sẽ nặng về tính toán hơn.

Câu trả lời dài: chúng ta có thể áp dụng K-means clustering. Với rất nhiều patches thu được, giả sử ta muốn xây dựng một *codebook* với chỉ khoảng 1000 *words*. Vậy thì ta cho $k = 1000$ rồi thực hiện K-means clustering trên toàn bộ số patches thu được (từ tập training). Sau khi thực hiện K-means clustering, ta thu được 1000 clusters và 1000 centers tương ứng. Mỗi centers này được coi là một *words*, và tất cả những điểm rơi vào cùng một cluster được coi là cùng một bag. Với ảnh trong tập test data, ta cũng lấy các patches rồi xem chúng rơi vào những bags nào. Từ đó suy ra vector đặc trưng cho mỗi bức ảnh. Chú ý rằng với $k = 1000$, mỗi bức ảnh sẽ được *mô tả* bởi một vector có số chiều 1000, tức là mỗi điểm dữ liệu bây giờ đã có số chiều bằng nhau, mặc dù ảnh thô đầu vào có thể có kích thước khác nhau.

11.3.6 Feature Scaling and Normalization

(Tham khảo Feature Scaling wiki).

Các điểm dữ liệu đôi khi được đo đạc với những đơn vị khác nhau, m 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.

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:

Rescaling

Phương pháp đơn giản nhất là đưa tất cả các thành phần về cùng một khoảng, $[0, 1]$ hoặc $[-1, 1]$ chẳng hạn, tùy thuộc vào ứng dụng. Nếu muốn đưa một thành phần (feature) về khoảng $[0, 1]$, công thức sẽ là:

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

trong đó x là giá trị ban đầu, x' là giá trị sau khi chuẩn hóa. $\min(x), \max(x)$ được tính trên toàn bộ dữ liệu training data ở cùng một thành phần. Việc này được thực hiện trên từng thành phần của vector dữ liệu $\mathbf{x}$.

Standardization

Một phương pháp nữa cũng hay được sử dụng là giả sử mỗi thành phần đều có phân phối chuẩn với kỳ vọng là 0 và phương sai là 1. Khi đó, công thức chuẩn hóa sẽ là:

$$x' = \frac{x - \bar{x}}{\sigma}$$

với $\bar{x}, \sigma$ lần lượt là kỳ vọng và phương sai (standard deviation) của thành phần đó trên toàn bộ training data.

Scaling to unit length

Một lựa chọn khác nữa cũng được sử dụng rộng rãi là chuẩn hóa các thành phần của mỗi vector dữ liệu sao cho toàn bộ vector có độ lớn (Ecluid, tức norm 2) bằng 1. Việc này có thể được thực hiện bằng:

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

11.4 Thảo luận

Xem ra thế giới Machine Learning rất rộng lớn và có rất nhiều thứ chúng ta cần làm. Và vẫn có khá nhiều thứ tôi có thể viết được. Tuy nhiên, blog này sẽ không tập trung nhiều vào Feature Learning, mặc dù sẽ có một vài bài nói về Dimensionality Reduction. Tôi sẽ sử dụng các bộ dữ liệu có sẵn, và đã qua bước Feature Learning.

11.5 Tài liệu tham khảo

[1] Feature Enginieering - wiki

[2] Feature Scaling wiki

[3] Csurka, Gabriella, et al. "Visual categorization with bags of keypoints." Workshop on statistical learning in computer vision, ECCV. Vol. 1. No. 1-22. 2004.

[4] Bag of Words model - wiki

[5] Bag of Words Meets Bags of Popcorn

12

Binary Classifiers cho các bài toán Classification

Cho tới bây giờ, ngoài *thuật toán lười* K-nearest neighbors, tôi đã giới thiệu với bạn đọc hai thuật toán cho các bài toán Classification: Perceptron Learning Algorithm và Logistic Regression. Hai thuật toán này được xếp vào loại Binary Classifiers vì chúng được xây dựng dựa trên ý tưởng về các bài toán classification với chỉ hai classes. Trong bài viết này, tôi sẽ cùng các bạn làm một vài ví dụ nhỏ về ứng dụng đơn giản (nhưng thú vị) của các binary classifiers, và cách mở rộng chúng để áp dụng cho các bài toán với nhiều classes (multi-class classification problems).

Vì Logistic Regression chỉ yêu cầu các classes là *nearly linearly separable* (tức có thể có vài điểm làm phá vỡ tính linear separability), tôi sẽ sử dụng Logistic Regression để đại diện cho các binary classifiers. *Chú ý rằng, có rất nhiều các thuật toán cho binary classification nữa mà tôi chưa giới thiệu. Tạm thời, với những gì đã viết, tôi chỉ sử dụng Logistic Regression cho các ví dụ với code mẫu. Các kỹ thuật trong bài viết này hoàn toàn có thể áp dụng cho các binary classifiers khác.*

12.1 Bài toán phân biệt giới tính dựa trên ảnh khuôn mặt

Chúng ta cùng bắt đầu với bài toán phân biệt giới tính dựa trên ảnh khuôn mặt. Về ảnh khuôn mặt, bộ cơ sở dữ liệu AR Face Database được sử dụng rộng rãi.

Bộ cơ sở dữ liệu này bao gồm hơn 4000 ảnh màu tương ứng với khuôn mặt của 126 người (70 nam, 56 nữ). Với mỗi người, 26 bức ảnh được chụp ở các điều kiện ánh sáng khác nhau, sắc thái biểu cảm khuôn mặt khác nhau, và bị che mắt (bởi kính râm) hoặc miệng (bởi khăn); và được chụp tại hai thời điểm khác nhau cách nhau 2 tuần.

Để cho đơn giản, tôi sử dụng bộ cơ sử AR Face thu gọn (có thể tìm thấy trong cùng trang web phía trên, mục *Other (relevant) downloads*). Bộ cơ sở dữ liệu thu gọn này bao gồm 2600 bức ảnh từ 50 nam và 50 nữ. Hơn nữa, các khuôn mặt cũng đã được xác định chính xác và được *cropped* với kích thước 165 x 120 (pixel) bằng phương pháp được mô tả trong bài báo PCA veus LDA. Tôi xin bỏ qua phần xử lý này và trực tiếp sử dụng ảnh đã cropped như một số ví dụ dưới đây:

Lưu ý:

- *Vì lý do bản quyền, tôi không được phép chia sẻ với các bạn bộ dữ liệu này. Các bạn muốn sở hữu có thể liên lạc với tác giả như hướng dẫn ở trong website AR Face Database. Một khi các bạn đã có tài khoản để download, tôi mong các bạn tôn trọng tác giả và không chia sẻ trực tiếp với bạn bè.*

- *Có một cách đơn giản và nhanh hơn để lấy được các feature vector (sau bước Feature Engineering) của cơ sở dữ liệu này mà không cần liên lạc với tác giả. Các bạn có thể tìm tại đây, phần* ***Downloads****, mục* ***Random face features for AR database****.*

Mỗi bức ảnh trong AR Face thu gọn được đặt tên dưới dạng `G-xxx-yy.bmp` Trong đó: `G` nhận một trong hai giá trị `M` (man) hoặc `W` (woman); `xxx` là id của người, nhận gía trị từ `001` đến `050`; `yy` là điều kiện chụp, nhận giá trị từ `01` đến `26`, trong đó các điều kiện có số thứ tự từ `01` đến `07` và từ `14` đến `20` là các khuôn mặt không bị che bởi kính hoặc khăn. Tôi tạm gọi mỗi *điều kiện* này là một *view*.

Để làm ví dụ cho thuật toán Logistic Regression, tôi lấy ảnh của 25 nam và 25 nữ đầu tiên làm tập training set; 25 nam và 25 nữ còn lại làm test set. Với mỗi người, tôi chỉ lấy các khuôn mặt không bị che bởi kính và khăn.

Feature Extraction: vì mỗi bức ảnh có kích thước `3x165x120` (số channels `3`, chiều cao `165`, chiều rộng `120`) là một số khá lớn nên ta sẽ làm thực hiện Feature Extraction bằng hai bước đơn giản sau (*bạn đọc được khuyến khích đọc bài Giới thiệu về Feature Engineering*):

- Chuyển ảnh màu về ảnh xám theo công thức `Y' = 0.299 R + 0.587 G + 0.114 B` (Xem thêm tại Grayscale - wiki).

- *Kéo dài* ảnh xám thu được thành 1 vector hàng có số chiều `165x120`, sau đó sử dụng một *random projection matrix* để giảm số chiều về `500`. Bạn đọc có thể thay giá trị này bằng các số khác nhỏ hơn `1000`.

Chúng ta có thể bắt đầu làm việc với Python ngay bây giờ. Tôi sẽ sử dụng hàm sklearn.linear_model.Logistic trong thư viện `sklearn` cho các ví dụ trong bài này. Nếu không muốn đọc phần này, bạn có thể lấy source code ở đây.

Chú ý: Hàm sklearn.linear_model.LogisticRegression nhận dữ liệu ở dạng vector hàng.

12.1.1 Làm việc với Python

Khai báo thư viện

```
import numpy as np
from sklearn import linear_model          # for logistic regression
```

```
from sklearn.metrics import accuracy_score # for evaluation
from scipy import misc                     # for loading image
np.random.seed(1)                          # for fixing random values
```

Phân chia training set và test set, lựa chọn các *views*.

```
path = '../data/AR/' # path to the database
train_ids = np.arange(1, 26)
test_ids = np.arange(26, 50)
view_ids = np.hstack((np.arange(1, 8), np.arange(14, 21)))
```

Tạo *random projection matrix*.

```
D = 165*120 # original dimension
d = 500 # new dimension

# generate the projection matrix
ProjectionMatrix = np.random.randn(D, d)
```

Xây dựng danh sách các tên files.

```
def build_list_fn(pre, img_ids, view_ids):
    """
    INPUT:
        pre = 'M-' or 'W-'
        img_ids: indexes of images
        view_ids: indexes of views
    OUTPUT:
        a list of filenames
    """
    list_fn = []
    for im_id in img_ids:
        for v_id in view_ids:
            fn = path + pre + str(im_id).zfill(3) + '-' + \
                str(v_id).zfill(2) + '.bmp'
            list_fn.append(fn)
    return list_fn
```

Feature Extraction: Xây dựng dữ liệu cho training set và test set.

```
def rgb2gray(rgb):
#     Y' = 0.299 R + 0.587 G + 0.114 B
    return rgb[:,:,0]*.299 + rgb[:, :, 1]*.587 + rgb[:, :, 2]*.114

# feature extraction
def vectorize_img(filename):
    # load image
    rgb = misc.imread(filename)
    # convert to gray scale
    gray = rgb2gray(rgb)
    # vectorization each row is a data point
    im_vec = gray.reshape(1, D)
    return im_vec

def build_data_matrix(img_ids, view_ids):
    total_imgs = img_ids.shape[0]*view_ids.shape[0]*2
```

```
    X_full = np.zeros((total_imgs, D))
    y = np.hstack((np.zeros((total_imgs/2, )), np.ones((total_imgs/2, ))))

    list_fn_m = build_list_fn('M-', img_ids, view_ids)
    list_fn_w = build_list_fn('W-', img_ids, view_ids)
    list_fn = list_fn_m + list_fn_w

    for i in range(len(list_fn)):
        X_full[i, :] = vectorize_img(list_fn[i])

    X = np.dot(X_full, ProjectionMatrix)
    return (X, y)

(X_train_full, y_train) = build_data_matrix(train_ids, view_ids)
x_mean = X_train_full.mean(axis = 0)
x_var  = X_train_full.var(axis = 0)

def feature_extraction(X):
    return (X - x_mean)/x_var

X_train = feature_extraction(X_train_full)
X_train_full = None ## free this variable

(X_test_full, y_test) = build_data_matrix(test_ids, view_ids)
X_test = feature_extraction(X_test_full)
X_test_full = None
```

Chú ý: Trong đoạn code trên tôi có sử dụng phương pháp chuẩn hóa dữ liệu Standardization. Trong đó `x_mean` và `x_var` lần lượt là vector kỳ vọng và phương sai của toàn bộ dữ liệu training. `X_train_full`, `X_test_full` là các ma trận dữ liệu đã được giảm số chiều nhưng chưa được chuẩn hóa. Hàm `feature_extraction` giúp chuẩn hóa dữ liệu dựa vào `x_mean` và `x_var` của `X_train_full`.

Đoạn code dưới đây thực hiện thuật toán Logistic Regression, dự đoán output của test data và đánh giá kết quả. Một chú ý nhỏ, hàm Logistic Regression trong thư viện sklearn có nhiều biến thể khác nhau. Để sử dụng thuật toán Logistic Regression *thuần* mà tôi đã giới thiệu trong bài Logistic Regression, chúng ta cần đặt giá trị cho `C` là một số lớn (để nghịch đảo của nó gần với 0. Tạm thời các bạn chưa cần quan tâm tới điều này, chỉ cần chọn `C` lớn là được).

```
logreg = linear_model.LogisticRegression(C=1e5) # just a big number
logreg.fit(X_train, y_train)

y_pred = logreg.predict(X_test)
print "Accuracy: %.2f %%" %(100*accuracy_score(y_test, y_pred))
```

Accuracy: 90.33

90.33

Để xác định *nhãn* của một ảnh, đầu ra của hàm sigmoid được so sánh với 0.5. Nếu giá trị đó lớn hơn 0.5, ta kết luận đó là ảnh của nam, ngược lại, đó là ảnh của nữ. Để xem giá trị

sau hàm sigmoid (tức xác suất để ảnh đó là nam), chúng ta sử dụng hàm `predict_proba` như sau:

```
def feature_extraction_fn(fn):
    """
    extract feature from filename
    """
    # vectorize
    im = vectorize_img(fn)
    # project
    im1 = np.dot(im, ProjectionMatrix)
    # standardization
    return feature_extraction(im1)

fn1 = path + 'M-036-18.bmp'
fn2 = path + 'W-045-01.bmp'
fn3 = path + 'M-048-01.bmp'
fn4 = path + 'W-027-02.bmp'

x1 = feature_extraction_fn(fn1)
p1 = logreg.predict_proba(x1)
print(p1)

x2 = feature_extraction_fn(fn2)
p2 = logreg.predict_proba(x2)
print(p2)

x3 = feature_extraction_fn(fn3)
p3 = logreg.predict_proba(x3)
print(p3)

x4 = feature_extraction_fn(fn4)
p4 = logreg.predict_proba(x4)
print(p4)
```

[[0.87940218 0.12059782]] [[0.0172217 0.9827783]] [[0.30458761 0.69541239]] [[0.83989242 0.16010758]]

Kết quả thu được là xác suất để bức ảnh đó là ảnh của nam (cột thứ nhất) và của nữ (cột thứ hai). Dưới đây là hình minh họa:

Hàng trên gồm các hình được phân loại đúng, hàng dưới gồm các hình bị phân loại sai. Có một vài nhận xét về hàng dưới. Từ hai bức ảnh hàng dưới, chúng ta có thể đoán rằng Logistic Regression quan tâm đến tóc phía sau gáy nhiều hơn là râu! Việc thuật toán dựa trên những đặc trưng nào của mỗi class phụ thuộc rất nhiều vào training data. Nếu trong training data, hầu hết nam không có râu và hầu hết nữ có tóc dài thì kết quả này là có thể lý giải được.

Trong Machine Learning, thuật toán là quan trọng, nhưng thuật toán tốt mà dữ liệu không tốt thì sẽ dẫn đến những tác dụng ngược!

(Source code cho ví dụ này có thể tìm thấy ở đây.)

12.2 Bài toán phân biệt hai chữ số viết tay

Chúng ta cùng sang ví dụ thứ hai về phân biệt hai chữ số trong bộ cơ sở dữ liệu MNIST. Cụ thể, tôi sẽ làm việc với hai chữ số 0 và 1. Bạn đọc hoàn toàn có thể thử với các chữ số khác bằng cách thay đổi một dòng lệnh. Khác với AR Face, bộ dữ liệu này có thể dễ dàng được download về từ trang chủ của nó.

Chúng ta có thể bắt tay vào làm luôn.

Khai báo thư viện:

```
# %reset
import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import accuracy_score
from display_network import *
```

Load toàn bộ dữ liệu:

```
mntrain = MNIST('../MNIST/')
mntrain.load_training()
Xtrain_all = np.asarray(mntrain.train_images)
ytrain_all = np.array(mntrain.train_labels.tolist())

mntest = MNIST('../MNIST/')
mntest.load_testing()
Xtest_all = np.asarray(mntest.test_images)
ytest_all = np.array(mntest.test_labels.tolist())
```

Sau bước này, toàn bộ dữ liệu training data và test data được lưu ở hai ma trận `X_train_all` và `X_test_all`, mỗi hàng của các ma trận này chứa một điểm dữ liệu, tức một bức ảnh đã được *vector hóa*.

Để lấy các hàng tương ứng với chữ số 0 và chữ số 1, ta khai báo biến sau:

```
cls = [[0], [1]]
```

Nếu bạn muốn thử với cặp `3` và `4`, chỉ cần thay dòng này bằng `cls = [[3], [4]]`. Nếu bạn muốn phân loại `(4, 7)` và `(5, 6)`, chỉ cần thay dòng này bằng `cls = [[4, 7], [5, 6]]`. Các cặp bất kỳ khác đều có thể thực hiện bằng cách thay chỉ một dòng này.

Đoạn code dưới đây thực hiện việc *extract* toàn bộ dữ liệu cho các chữ số `0` và `1` trong tập training data và test data.

```
def extract_data(X, y, classes):
    """
    X: numpy array, matrix of size (N, d), d is data dim
    y: numpy array, size (N, )
    cls: two lists of labels. For example:
        cls = [[1, 4, 7], [5, 6, 8]]
```

```
    return:
        X: extracted data
        y: extracted label
            (0 and 1, corresponding to two lists in cls)
    """
    y_res_id = np.array([])
    for i in cls[0]:
        y_res_id = np.hstack((y_res_id, np.where(y == i)[0]))
    n0 = len(y_res_id)

    for i in cls[1]:
        y_res_id = np.hstack((y_res_id, np.where(y == i)[0]))
    n1 = len(y_res_id) - n0

    y_res_id = y_res_id.astype(int)
    X_res = X[y_res_id, :]/255.0
    y_res = np.asarray([0]*n0 + [1]*n1)
    return (X_res, y_res)


# extract data for training
(X_train, y_train) = extract_data(Xtrain_all, ytrain_all, cls)

# extract data for test
(X_test, y_test) = extract_data(Xtest_all, ytest_all, cls)
```

Vì mỗi điểm dữ liệu có số phần tử là 784 (28x28), là một số khá nhỏ, nên ta không cần thêm bước giảm số chiều dữ liệu nữa. Tuy nhiên, tôi có thực hiện thêm một bước chuẩn hóa để đưa dữ liệu về đoạn `[0, 1]` bằng cách chia toàn bộ hai ma trận dữ liệu cho `255.0`.

Tới đây ta có thể *train* mô hình Logistic Regression và đánh giá mô hình này.

```
# train the logistic regression model
logreg = linear_model.LogisticRegression(C=1e5) # just a big number
logreg.fit(X_train, y_train)

# predict
y_pred = logreg.predict(X_test)
print "Accuracy: %.2f %%" %(100*accuracy_score(y_test, y_pred.tolist()))
```

Accuracy: 99.95

Tuyệt vời, gần như 100

Chúng ta cùng đi tìm những ảnh bị phân loại sai:

```
# display misclassified image(s)
mis = np.where((y_pred - y_test) != 0)[0]
Xmis = X_test[mis, :]

plt.axis('off')
A = display_network(Xmis.T)
f2 = plt.imshow(A, interpolation='nearest' )
plt.gray()
plt.show()
```

Như vậy là chỉ có một ảnh bị phân loại sai. Ảnh này là chữ số 0 nhưng bị misclassified thành chữ số 1, có thể vì nét đậm nhất của nó rất giống với chữ số 1.

Source code cho ví dụ này có thể được tìm thấy ở đây.

12.3 Binary Classifiers cho Multi-class Classification problems

Có lẽ nhiều bạn đang đặt câu hỏi: Các ví dụ trên đây đều làm với bài toán có hai classes. Vậy nếu có nhiều hơn hai classes, ví dụ như 10 classes của MNIST, thì làm thế nào?

Có nhiều thuật toán khác được xây dựng riêng cho các bài toán với nhiều classes (multi-class classification problems), tôi sẽ giới thiệu sau. Còn bây giờ, chúng ta vẫn có thể sử dụng các *binary classifiers* để thực hiện công việc này, với một chút thay đổi.

Có *ít nhất* bốn cách để áp dụng *binary classifiers* vào các bài toán multi-class classification:

12.3.1 One-vs-one

Xây dựng rất nhiều bộ binary classifiers cho từng cặp classes. Bộ thứ nhất phân biệt class 1 và class 2, bộ thứ hai phân biệt class 1 và class 3, ... Khi có một dữ liệu mới vào, đưa nó vào toàn bộ các bộ binary classifiers trên. Kết quả cuối cùng có thể được xác định bằng cách xem class nào mà điểm dữ liệu đó được phân vào nhiều nhất (major voting). Hoặc với Logistic Regression thì ta có thể tính *tổng các xác suất* tìm được sau mỗi bộ binary classifier.

Như vậy, nếu có C classes thì tổng số binary classifiers phải dùng là $\frac{n(n-1)}{2}$. 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 tìm được (là `5` hoặc `6`) đều không hợp lý!

12.3.2 Hierarchical (phân tầng)

Các làm như **one-vs-one** sẽ mất rất nhiều thời gian training vì có quá nhiều bộ phân lớp cần được xây dựng. Một cách khác giúp *tiết kiệm* số binary classifiers hơn đó là **hierarchical**. Ý tưởng như sau:

Ví dụ với MNIST với 4 chữ số `4, 5, 6, 7`. Vì ta thấy chữ số `4` và `7` khá giống nhau, chữ số `5` và `6` khá giống nhau nên trước tiên chúng 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` nữa. Tổng cộng, ta cần 3 bộ binary classifiers. 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 phương pháp này là sử dụng ít bộ binary classifiers hơn **one-vs-one**. Hạn chế lớn nhất của nó là việc nếu chỉ một binary classifier cho kết quả sai thì kết quả cuối cùng chắc chắn sẽ sai. Ví dụ, nếu 1 ảnh chứa chữ số `5`, nhưng ngay bước đầu tiên đã bị misclassifed sang nhánh `[4, 7]` thì kết quả cuối cùng sẽ là `4` hoặc `7`, cả hai đều sai.

12.3.3 Binary coding

Có một cách giảm số binary classifiers hơn nữa là **binary coding**, tức *mã hóa* output của mỗi class bằng một số nhị phân. Ví dụ, nếu có 4 classes thì class thứ nhất được mã hóa là `00`, ba class kia được mã hóa lần lượt là `01, 10` và `11`. Với cách làm này, số bộ binary classifiers phải thực hiện chỉ là $m = \lceil \log_2(C) \rceil$ trong đó C là số lượng class, $\lceil a \rceil$ là *số nguyên nhỏ nhất không nhỏ hơn* a. Class thứ nhất sẽ đi tìm bit đầu tiên của output (đã được mã hóa nhị phân), class thứ hai sẽ đi tìm bit thứ hai, ...

Cách làm này sử dụng một số lượng nhỏ nhất các bộ *binary classifiers*. Nhưng 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 classes không phải là lũy thừa của hai, mã nhị phân nhận được có thể là một giá trị không tương ứng với class nào!

12.3.4 one-vs-rest hay one-hot coding

Phương pháp được sử dụng nhiều nhất là **one-vs-rest** (một số tài liệu gọi là **ove-vs-all**, **one-against-rest**, hoặc **one-against-all**) . Cụ thể, nếu có C classes thì ta sẽ xây dựng C classifiers, mỗi classifier tương ứng với một class. Classifier thứ nhất giúp phân biệt **`class 1`** vs **`not class 1`**, tức xem một điểm có thuộc class 1 hay không, hoặc xác suất để một điểm rơi vào class 1 là bao nhiêu. Tương tự như thế, classifier thứ hai sẽ phân biệt **`class 2`** vs **`not class 2`**, ... Kết quả cuối cùng có thể được xác định bằng cách xác định class mà một điểm rơi vào với xác suất cao nhất.

Phương pháp này còn được gọi là **one-hot coding** (được sử dụng nhiều nên có rất nhiều tên) vì với cách mã hóa trên, giả sử có 4 classes, class 1, 2, 3, 4 sẽ lần lượt được mã hóa dưới dạng nhị phân bởi `1000, 0100, 0010` hoặc `0001`. One-hot vì chỉ có *one* bit là *hot* (bằng `1`).

Hàm Logistic Regression trong thư viện sklearn có thể được dùng trực tiếp để áp dụng vào các bài toán multi-class classification với phương pháp **one-vs-rest**. Với bài toán MNIST như nêu ở phần 2, ta có thể thêm ba dòng lệnh sau để chạy trên toàn bộ 10 classes:

```
logreg.fit(Xtrain_all, ytrain_all)
y_pred = logreg.predict(Xtest_all)
print "Accuracy: %.2f %%" %(100*accuracy_score(ytest_all, y_pred.tolist()))
```

Kết quả thu được khoảng 91% sau hơn 20 phút chạy (tùy thuộc vào máy). Đây vẫn là một kết quả quá thấp so với con số 99.7%. Thậm chí phương pháp học máy *không học gì* như K-neareast neighbors cũng đã đạt hơn 96% với thời gian chạy ngắn hơn một chút.

Một chú ý nhỏ: phương pháp mặc định cho các bài toán multi-class của hàm này được xác định bởi biến `multi_class`. Có hai lựa chọn cho biến này, trong đó lựa chọn mặc định là `ovr` tức **one-vs-rest**, lựa chọn còn lại sẽ được tôi đề cập trong một bài gần đây. Lựa chọn thứ hai không phải cho binary classifiers nên tôi không đề cập trong bài này, có thể sau một vài bài nữa (Xem thêm `sklearn.linear_model.LogisticRegression`)

12.4 Thảo luận

12.4.1 Kết hợp các phương pháp trên

Nhắc lại rằng các linear binary classifiers tôi đã trình bày yêu cầu dữ liệu là *linearly separable* hoặc *nearly linearly separable*. Ta cũng có thể mở rộng định nghĩa này cho các bài toán multi-class. Nếu hai class bất kỳ là *linearly separable* thì ta coi dữ liệu đó là *linearly separable*.

Thế nhưng, có những loại dữ liệu *linearly separable* mà chỉ một số trong 4 phương pháp trên đây là phù hợp, hoặc có những loại dữ liệu yêu cầu phải kết hợp nhiều phương pháp mới thực hiện được. Xét ba ví dụ sau:

- Hình 4a): cả 4 phương pháp trên đây đều có thể áp dụng được.
- Hình 4b): one-vs-rest không phù hợp vì class màu xanh lục và class *rest* (hợp của xanh lam và đỏ) là không *linearly separable*. Lúc này, one-vs-one hoặc hierarchical phù hợp hơn.
- Hình 4c): Tương tự như trên, ba class lam, lục, đỏ thẳng hàng nên sẽ không dùng được one-vs-rét. one-vs-one vẫn làm việc vì từng đôi class một 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* vs *không đỏ*. Nếu một điểm dữ liệu là *không đỏ*, với 3 class còn lại, chúng ta lại quay lại trường hợp Hình 4a) và có thể dùng các phương pháp khác. Nhưng khó khăn vẫn nằm ở việc phân nhóm như thế nào, liệu rằng những class nào có thể cho vào cùng một nhóm? Với những dữ liệu đơn giản, K-means clustering có thể là một giải pháp!

Bạn đọc có thể xem thêm ví dụ áp dụng Logistic Regression cho cơ sở dữ liệu Iris trong link này

12.4.2 Biểu diễn dưới dạng Neural Networks

Lấy ví dụ với bài toán có 4 classes 1, 2, 3, 4; ta có thể biểu diễn các mô hình được đề cập trong phần 3 dưới dạng sau đây (giả sử input có số chiều là 7 và node output màu đỏ biểu diễn chung cho cả PLA, Logistic Regression và các networks với activation function khác):

Lúc này, thay vì chỉ có 1 node output như các phương pháp tôi đề cập trước đây (Linear Regression, Perceptron Learning Algorithm, Logistic Regression), chúng ta thấy rằng các networks này đều có nhiều outputs. 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 binary classifiers trong mỗi network cũng được tổng quát lên nhớ các phép tính với ma trận.

Lấy ví dụ với công thức cập nhật của logistic sigmoid regression :

$$\mathbf{w} = \mathbf{w} + \eta(y_i - z_i)\mathbf{x}_i$$

Có thể tổng quát thành:

$$\mathbf{W} = \mathbf{W} + \eta\mathbf{x}_i(\mathbf{y}_i - \mathbf{z}_i)^T$$

Với $\mathbf{W}, \mathbf{y}_i, \mathbf{z}_i$ lần lượt là ma trận trọng số, vector (cột) output *thật* với toàn bộ các binary classifiers 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 với Logistic Regression, vector $\mathbf{y}_i$ là một binary vector, vector $\mathbf{z}_i$ gồm các phần tử nằm trong khoảng $(0, 1)$.

12.4.3 Hạn chế của one-vs-rest

Xem xét lại phương pháp one-vs-rest theo góc nhìn xác suất, một điểm dữ liệu có thể được dự đoán thuộc vào class $1, 2, \ldots, C$ với xác suất lần lượt là $p_1, p_2, \ldots, p_C$. Tuy nhiên, tổng các xác suất này có thể không bằng 1! Có một phương pháp có thể làm cho nó *hợp lý hơn*, tức *ép* tổng các xác suất này bằng 1. Khi đó, với 1 điểm dữ liệu ta có thể nói xác suất nó rơi vào mỗi class là bao nhiêu. Phương pháp hấp dẫn này sẽ được đề cập trong bài Softmax Regression. Mời bạn đón đọc.

12.5 Tài liệu tham khảo

[1] Multiclass classification - wiki

[2] Logistic Regression 3-class Classifier

13

Softmax Regression

—

Các bài toán classification thực tế thường có rất nhiều classes (multi-class), các binary classifiers mặc dù có thể áp dụng cho các bài toán multi-class, chúng vẫn có những hạn chế nhất định. Với binary classifiers, kỹ thuật được sử dụng nhiều nhất **one-vs-rest** có một hạn chế về tổng các xác suất. Trong post này, một phương pháp mở rộng của Logistic Regression sẽ được giới thiệu giúp khắc phục hạn chế trên. Một lần nữa, dù là Softmax **Regression**, phương pháp này được sử dụng rộng rãi như một phương pháp classification.

Một lưu ý nhỏ: Hàm mất mát của Softmax Regression trông có vẻ khá phức tạp, nhưng nếu kiên trì đọc đến phần phương pháp tối ưu, các bạn sẽ thấy vẻ đẹp ẩn sau sự phức tạp đó. Gradient của hàm mất mát và công thức cập nhật ma trận trọng số là rất đơn giản. (Đơn giản sau vài bước biến đổi toán học *trông có vẻ* phức tạp).

Nếu có điểm nào khó hiểu, bạn đọc được khuyến khích đọc lại các bài trước, trong đó quan trọng nhất là Bài 10: Logistic Regression.

13.1 Giới thiệu

Tôi xin phép được bắt đầu từ mô hình **one-vs-rest** được trình bày trong bài trước. Output layer (màu đỏ nhạt) có thể phân tách thành hai *sublayer* như hình dưới đây:

Dữ liệu $\mathbf{x}$ có số chiều là $(d+1)$ vì có phần tử 1 được thêm vào phía trước, thể hiện hệ số tự do trong hàm tuyến tính. Hệ số tự do w_{0j} còn được gọi là bias.

Giả sử số classes là C. Với one-vs-rest, chúng ta cần xây dựng C Logistic Regression khác nhau. Các *đầu ra dự đoán* được tính theo hàm sigmoid:

$$a_i = \text{sigmoid}(z_i) = \text{sigmoid}(\mathbf{w}_i^T \mathbf{x})$$

Trong kỹ thuật này, các phần tử $a_i, i = 1, 2, \ldots, C$ được suy ra trực tiếp chỉ với z_i. Vì vậy, không có mối quan hệ chặt chẽ nào giữa các a_i, tức tổng của chúng có thể nhỏ hơn hoặc

lớn hơn 1. Nếu ta có thể khai thác được mỗi quan hệ giữa các z_i thì kết quả của bài toán classification có thể tốt hơn.

Chú ý rằng các mô hình Linear Regression, PLA, Logistic Regression chỉ có 1 node ở output layer. Trong các trường hợp đó, tham số mô hình chỉ là 1 vector $\mathbf{w}$. Trong trường hợp output layer có nhiều hơn 1 node, tham số mô hình sẽ là tập hợp tham số $\mathbf{w}_i$ ứng với từng node. Lúc này, ta có *ma trận trọng số* $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_C]$.

13.2 Softmax function

13.2.1 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 class i. Vậy điều kiện cần là các a_i phải dương và tổng của chúng bằng 1. Để có thể thỏa mãn điều kiện này, chúng ta cần *nhìn vào* mọi giá trị z_i và dựa trên quan hệ giữa các z_i này để tính toán giá trị của a_i. Ngoài các điều kiện a_i lớn hơn 0 và có tổng bằng 1, chúng 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 class i càng cao. Điều kiện cuối này chỉ ra rằng chúng ta cần một hàm đồng biến ở đây.

Chú ý rằng z_i có thể nhận giá trị cả âm và dương. Một hàm số *mượt* 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 *mượt* để thuận lợi hơn trong việc tính đạo hàm sau này. Điều kiện cuối cùng, tổng các a_i bằng 1 có thể được đảm bảo nếu:

$$a_i = \frac{\exp(z_i)}{\sum_{j=1}^{C}\exp(z_j)}, \quad \forall i = 1, 2, \ldots, C$$

Hàm số này, tính tất cả các a_i dựa vào tất cả các z_i, thõa mãn tất cả các điều kiện đã xét: dương, tổng bằng 1, giữ được *thứ tự* của z_i. Hàm số này được gọi là *softmax function*. Chú ý rằng với cách định nghĩa này, không có xác suất a_i nào tuyệt đối bằng 0 hoặc tuyệt đối bằng 1, mặc dù chúng có thể rất gần 0 hoặc 1 khi z_i rất nhỏ hoặc rất lớn khi so sánh với các $z_j, j \neq i$.

Lúc này, ta có thể giả sử rằng:

$$P(y_k = i|\mathbf{x}_k; \mathbf{W}) = a_i$$

Trong đó, $P(y = i\|\mathbf{x}; \mathbf{W})$ được hiểu là xác suất để một điểm dữ liệu $\mathbf{x}$ rơi vào class thứ i nếu biết tham số mô hình (ma trận trọng số) là $\mathbf{W}$.

Hình vẽ dưới đây thể hiện mạng Softmax Regression dưới dạng neural network:

Ở phần bên phải, hàm tuyến tính Σ và hàm softmax (activation function) được tách riêng ra để phục vụ cho mục đích minh họa. Dạng *short form* ở bên phải là dạng hay được sử dụng trong các Neural Networks, lớp **a** được ngầm hiểu là bao gồm cả lớp **z**.

13.2.2 Softmax function trong Python

Dưới đây là một đoạn code viết hàm softmax. Đầu vào là một ma trận với mỗi cột là một vector **z**, đầu ra cũng là một ma trận mà mỗi cột có giá trị là $\mathbf{a} = \text{softmax}(\mathbf{z})$. Các giá trị của **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 score.
    """
    e_Z = np.exp(Z)
    A = e_Z / e_Z.sum(axis = 0)
    return A
```

13.2.3 Một vài ví dụ

Hình 3 dưới đây là 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 classes là 3. Hàng dưới màu đỏ nhạt thể hiện các giá trị đầu ra a_i của hàm softmax.

Có một vài quan sát như sau:

- Cột 1: Nếu các z_i bằng nhau, 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, nhưng các giá trị khác thay đổi, thì mặc dù xác suất tương ứng a_1 vẫn là lớn nhất, nhưng nó đã thay đổi lên hơn 0.5. Đây chính là một lý do mà tên của hàm này có từ *soft*. (*max* vì phẩn từ lớn nhất vẫn là phần tử lớn nhất).
- 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 trong link này, kéo xuống phần Softmax.

13.2.4 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 sau:

$$\begin{aligned}\frac{\exp(z_i)}{\sum_{j=1}^{C}\exp(z_j)} &= \frac{\exp(-c)\exp(z_i)}{\exp(-c)\sum_{j=1}^{C}\exp(z_j)} \\ &= \frac{\exp(z_i - c)}{\sum_{j=1}^{C}\exp(z_j - c)}\end{aligned}$$

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 cột của ma trận đầu vào `V` đi giá trị lớn nhất trong cột đó. Ta có phiên bản ổn định hơn là `softmax_stable`:

```
def softmax_stable(Z):
    """
    Compute softmax values for each sets of scores in V.
    each column of V is a set of score.
    """
    e_Z = np.exp(Z - np.max(Z, axis = 0, keepdims = True))
    A = e_Z / e_Z.sum(axis = 0)
    return A
```

trong đó `axis = 0` nghĩa là lấy **`max`** theo cột (`axis = 1` sẽ lấy max theo hàng), `keepdims = True` để đảm bảo phép trừ giữa ma trận `V` và vector thực hiện được.

13.3 Hàm mất mát và phương pháp tối ưu

13.3.1 One hot coding

Với cách biểu diễn network như trên, mỗi output sẽ không còn là một giá trị tương ứng với mỗi class nữa mà sẽ là một vector có đúng 1 phần tử bằng 1, các phần tử còn lại bằng 0. Phần tử bằng 1 nằm ở vị trí tương ứng với class đó, thể hiện rằng điểm dữ liệu đang xét rơi vào class này với xác suất bằng 1 (*sự thật* là như thế, không cần dự đoán). Cách *mã hóa* output này chính là *one-hot coding* mà tôi đã đề cập trong bài K-means clustering và bài trước.

Khi sử dụng mô hình Softmax Regression, với mỗi đầu vào $\mathbf{x}$, ta sẽ có *đầu ra dự đoán* là $\mathbf{a} = \text{softmax}(\mathbf{W}^T\mathbf{x})$. Trong khi đó, *đầu ra thực sự* chúng ta có là vector $\mathbf{y}$ được biểu diễn dưới dạng one-hot coding.

Hàm mất mát sẽ được xây dựng để tối thiểu sự khác nhau giữa *đầu ra dự đoán* $\mathbf{a}$ và *đầu ra thực sự* $\mathbf{y}$. Một lựa chọn đầu tiên ta có thể nghĩ tới là:

$$J(\mathbf{W}) = \sum_{i=1}^{N} \|\|\mathbf{a}_i - \mathbf{y}_i\|\|_2^2$$

Tuy nhiên đây chưa phải là một lựa chọn tốt. Khi đánh giá sự khác nhau (hay khoảng cách) giữa hai phân bố xác suất (probability distributions), chúng ta có một đại lượng đo đếm khác hiệu quả hơn. Đại lượng đó có tên là **cross entropy**.

13.3.2 Cross Entropy

Cross entropy giữa hai phân phối $\mathbf{p}$ và $\mathbf{q}$ được định nghĩa là:

$$H(\mathbf{p}, \mathbf{q}) = \mathbf{E}_{\mathbf{p}}[-\log \mathbf{q}]$$

Với $\mathbf{p}$ và $\mathbf{q}$ là rời rạc (như $\mathbf{y}$ và $\mathbf{a}$ trong bài toán của chúng ta), công thức này được viết dưới dạng:

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{C} p_i \log q_i \quad (1)$$

Để hiểu rõ hơn ưu điểm của hàm cross entropy và hàm bình phương khoảng cách thông thường, chúng ta cùng xem Hình 4 dưới đây. Dây là ví dụ trong trường hợp $C = 2$ và p_1 lần lượt nhận các giá trị $0.5, 0.1$ và 0.8.

Có hai nhận xét quan trọng sau đây:

- Giá trị nhỏ nhất của cả hai hàm số đạt được khi $q = p$ tại hoành độ của các điểm màu xanh lục.
- Quan trọng hơn, hàm cross entropy nhận giá trị rất cao (tức loss rất cao) khi q ở xa p. Trong khi đó, sự chênh lệch giữa các loss ở gần hay xa nghiệm của hàm bình phương khoảng cách $(q - p)^2$ là không đáng kể. 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.

Chú ý: 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 supervised learning, $\mathbf{p}$ thường là *đầu ra thực sự* vì đầu ra thực sự chỉ có 1 thành phần bằng 1, còn lại bằng 0 (one-hot), $\mathbf{q}$ thường là *đầu ra dự đoán*, khi mà không có xác suất nào tuyệt đối bằng 1 hoặc tuyệt đối bằng 0 cả.

Trong Logistic Regression, chúng ta cũng có hai phân phối đơn giản. (i) *Đầu ra thực sự* của điểm dữ liệu đầu vào $\mathbf{x}_i$ có phân phối xác suất là $[y_i; 1-y_i]$ với y_i là xác suất để điểm dữ liệu đầu vào rơi vào class thứ nhất (bằng 1 nếu $y_i = 1$, bằng 0 nếu $y_i = 0$). (ii). *Đầu ra dự đoán* của điểm dữ liệu đó là $a_i = \text{sigmoid}(\mathbf{w}^T\mathbf{x})$ là xác suất để điểm đó rơi vào class thứ nhất. Xác suất để điểm đó rơi vào class thứ hai có thể được dễ dàng suy ra lf $1 - a_i$. Vì vậy, hàm mất mát trong Logistic Regression:

$$J(\mathbf{w}) = -\sum_{i=1}^{N}(y_i \log a_i + (1-y_i)\log(1-a_i))$$

chính là một trường hợp đặc biệt của Cross Entropy. (N được dùng để thể hiện số điểm dữ liệu trong tập training).

Với Softmax Regression, trong trường hợp có C classes, *loss* giữa đầu ra dự đoán và đầu ra thực sự của một điểm dữ liệu $\mathbf{x}_i$ được tính bằng:

$$J(\mathbf{W}; \mathbf{x}_i, \mathbf{y}_i) = -\sum_{j=1}^{C} y_{ji} \log(a_{ji})$$

Với y_{ji} và a_{ji} lần lượt là 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 $\mathbf{W}$.

13.3.3 Hàm mất mát cho Softmax Regression

Kết hợp tất cả các cặp dữ liệu $\mathbf{x}_i, \mathbf{y}_i, i = 1, 2, \ldots, N$, chúng ta sẽ có hàm mất mát cho Softmax Regression như sau:

$$J(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ji} \log(a_{ji}) = -\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ji} \log\left(\frac{\exp(\mathbf{w}_j^T\mathbf{x}_i)}{\sum_{k=1}^{C}\exp(\mathbf{w}_k^T\mathbf{x}_i)}\right) \tag{13.1}$$

Với ma trận trọng số $\mathbf{W}$ là biến cần tối ưu. Hàm mất mát này trông *có vẻ đáng sợ*, nhưng đừng sợ, đọc tiếp các bạn sẽ thấy đạo hàm của nó rất đẹp (*và đáng yêu*).

13.3.4 Tối ưu hàm mất mát

Một lần nữa, chúng ta lại sử dụng Stochastic Gradient Descent (SGD) ở đây.

Với chỉ một cặp dữ liệu $(\mathbf{x}_i, \mathbf{y}_i)$, ta có:

$$J_i(\mathbf{W}) \triangleq J(\mathbf{W}; \mathbf{x}_i, \mathbf{y}_i) =$$

$$\begin{aligned}
&= -\sum_{j=1}^{C} y_{ji} \log \left(\frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^{C} \exp(\mathbf{w}_k^T \mathbf{x}_i)} \right) \\
&= -\sum_{j=1}^{C} \left(y_{ji} \mathbf{w}_j^T \mathbf{x}_i - y_{ji} \log \left(\sum_{k=1}^{C} \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \right) \\
&= -\sum_{j=1}^{C} y_{ji} \mathbf{w}_j^T \mathbf{x}_i + \log \left(\sum_{k=1}^{C} \exp(\mathbf{w}_k^T \mathbf{x}_i) \right) \quad (3)
\end{aligned}$$

trong biến đổi ở dòng cuối cùng, tôi đã sử dụng quan sát: $\sum_{j=1}^{C} y_{ji} = 1$ vì nó là tổng các xác suất.

Tiếp theo ta sử dụng công thức:

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}} = \left[\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_1}, \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_2}, \ldots, \frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_C} \right] \quad (4)$$

Trong đó, gradient theo từng cột có thể tính được dựa theo (3):

$$\begin{aligned}
\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{w}_j} &= -y_{ji} \mathbf{x}_i + \frac{\exp(\mathbf{w}_j^T \mathbf{x}_i)}{\sum_{k=1}^{C} \exp(\mathbf{w}_k^T \mathbf{x}_i)} \mathbf{x}_i \\
&= y_{ji} \mathbf{x}_i + a_{ji} \mathbf{x}_i = \mathbf{x}_i (a_{ji} - y_{ji}) \\
&= e_{ji} \mathbf{x}_i \text{ (where } e_{ji} = a_{ji} - y_{ji}) \quad (5)
\end{aligned}$$

Giá trị $e_{ji} = a_{ji} - y_{ji}$ có thể được coi là *sai số dự đoán*.

Đến đây ta đã được biểu thức rất đẹp rồi. Kết hợp (4) và (5) ta có:

$$\frac{\partial J_i(\mathbf{W})}{\partial \mathbf{W}} = \mathbf{x}_i [e_{1i}, e_{2i}, \ldots, e_{Ci}] = \mathbf{x}_i \mathbf{e}_i^T$$

Từ đây ta cũng có thể suy ra rằng:

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \sum_{i=1}^{N} \mathbf{x}_i \mathbf{e}_i^T = \mathbf{X} \mathbf{E}^T$$

với $\mathbf{E} = \mathbf{A} - \mathbf{Y}$. Công thức tính gradient đơn giản thế này giúp cho cả Batch Gradient Descent, Stochastic Gradient Descent (SGD), và Mini-batch Gradient Descent đều có thể dễ dàng được áp dụng.

Giả sử rằng chúng ta sử dụng SGD, công thức cập nhật cho ma trận trọng số $\mathbf{W}$ sẽ là:

$$\mathbf{W} = \mathbf{W} + \eta \mathbf{x}_i (\mathbf{y}_i - \mathbf{a}_i)^T$$

Bạn có thấy công thức này giống với công thức cập nhật của Logistic Regression không!

Thực ra:

13.3.5 Logistic Regression là một trường hợp đặt biệt của Softmax Regression

Khi $C = 2$, Softmax Regression và Logistic Regression là giống nhau. Thật vậy, đầu ra dự đoán của Softmax Regression với $C = 2$ có thể được viết dưới dạng:

$$a_1 = \frac{\exp(\mathbf{w}_1 \mathbf{x})}{\exp(\mathbf{w}_1^T \mathbf{x}) + \exp(\mathbf{w}_2^T \mathbf{x})} = \frac{1}{1 + \exp((\mathbf{w}_2 - \mathbf{w}_1)^T \mathbf{x})} \tag{13.2}$$

Đây chính là sigmoid function, là đầu ra dự đoán theo Logistic Regression. Khi $C = 2$, bạn đọc cũng có thể thấy rằng hàm mất mát của Logistic và Softmax Regression đều là cross entropy. Hơn nữa, mặc dù có 2 outputs, Softmax Regression có thể rút gọn thành 1 output vì tổng 2 outputs luôn luôn bằng 1.

Softmax Regression còn có các tên gọi khác là Multinomial Logistic Regression, Maximum Entropy Classifier, hay rất nhiều tên khác nữa. Xem thêm Multinomial logistic regression - Wikipedia

13.4 Một vài lưu ý khi lập trình với Python

13.4.1 Bắt đầu với dữ liệu nhỏ

Các bài toán Machine Learning thường có độ phức tạp cao với lượng dữ liệu lớn và nhiều chiều. Để có thể áp dụng một thuật toán vào một bài toán cụ thể, trước tiên chúng ta cần áp dụng thuật toán đó vào *simulated data* (dữ liệu giả) với số chiều và số điểm dữ liệu nhỏ hơn. *Simulated data* này thường được tạo ngẫu nhiên (có thể thêm vài ràng buộc tùy vào đặc thù của dữ liệu). Với *simulated data* nhỏ, chúng ta có thể debug nhanh hơn và thử với nhiều trường hợp *simulated data* khác nhau. Khi nào thấy thuật toán chạy đúng chúng ta mới đưa *dữ liệu thật* vào.

Với Softmax Regression, tôi tạo *simulated data* như sau:

```
import numpy as np

# randomly generate data
N = 2 # number of training sample
d = 2 # data dimension
C = 3 # number of classes

X = np.random.randn(d, N)
y = np.random.randint(0, 3, (N,))
```

Trong ví dụ đơn giản này, số điểm dữ liệu chỉ là `N = 2`, số chiều dữ liệu `d = 2`, và số classes `C = 3`. Những giá trị đủ nhỏ này giúp cho việc kiểm tra có thể được thực hiện một cách tức thì. Sau khi thuật toán chạy đúng với những giá trị nhỏ này, ta có thể thay `N, d, C` bằng vài giá trị khác trước khi sử dụng dữ liệu thật.

13.4.2 Ma trận one-hot coding

Có một bước quan trọng nữa trong Softmax Regression là phải chuyển đổi mỗi label y_i thành một vector $\mathbf{y}_i$ dưới dạng one-hot coding. Trong đó, chỉ có đúng một phần tử của $\mathbf{y}_i$ bằng 1, các phần tử còn lại bằng 0. Như vậy, với N điểm dữ liệu và C classes, chúng ta sẽ có một ma trận có kích thước $C \times N$ trong đó mỗi cột chỉ có đúng 1 phần tử bằng 1, còn lại bằng 0. Nếu chúng ta lưu toàn bộ dữ liệu này thì sẽ bị lãng phí bộ nhớ.

Một cách thường được sử dụng là lưu ma trận output $\mathbf{Y}$ dưới dạng *sparse matrix*. Về cơ bản, cách làm này chỉ lưu các **vị trí** khác 0 của ma trận và **giá trị** khác 0 đó.

Python có hàm scipy.sparse.coo_matrix giúp chúng ta thực hiện việc này. Với one-hot coding, tôi thực hiện như sau:

```
## One-hot coding
from scipy import sparse
def convert_labels(y, C = C):
    """
    convert 1d label to a matrix label: each column of this
    matrix coresponding to 1 element in y. In i-th column of Y,
    only one non-zeros element located in the y[i]-th position,
    and = 1 ex: y = [0, 2, 1, 0], and 3 classes then return

            [[1, 0, 0, 1],
             [0, 0, 1, 0],
             [0, 1, 0, 0]]
    """
    Y = sparse.coo_matrix((np.ones_like(y),
        (y, np.arange(len(y)))), shape = (C, len(y))).toarray()
    return Y
```

13.4.3 Kiểm tra đạo hàm

Điều cốt lõi trong cách tối ưu hàm mất mát là tính gradient. Với biểu thức toán trông *khá rối mắt* như trên, rất dễ để các bạn nhầm lẫn ở một bước nào đó. Softmax Regression vẫn là một thuật toán đơn giản, sau này các bạn sẽ thấy nhưng biểu thức phức tạp hơn nhiều. Rất khó để có thể tính toán đúng gradient ở ngay lần thử đầu tiên.

Trong thực nghiệm, một cách thường được làm là so sánh gradient tính được với *numeric gradient*, tức gradient tính theo định nghĩa. Bạn đọc được khuyến khích đọc cách Kiểm tra đạo hàm.

Việc kiểm tra đạo hàm được thực hiện như sau:

```
# cost or loss function
def cost(X, Y, W):
    A = softmax(W.T.dot(X))
    return -np.sum(Y*np.log(A))

W_init = np.random.randn(d, C)

def grad(X, Y, W):
    A = softmax((W.T.dot(X)))
    E = A - Y
    return X.dot(E.T)

def numerical_grad(X, Y, W, cost):
    eps = 1e-6
    g = np.zeros_like(W)
    for i in range(W.shape[0]):
        for j in range(W.shape[1]):
            W_p = W.copy()
            W_n = W.copy()
            W_p[i, j] += eps
            W_n[i, j] -= eps
            g[i,j] = (cost(X, Y, W_p) - cost(X, Y, W_n))/(2*eps)
    return g

g1 = grad(X, Y, W_init)
g2 = numerical_grad(X, Y, W_init, cost)

print(np.linalg.norm(g1 - g2))
```

2.70479295591e-10

Như vậy, sự khác biệt giữa hai đạo hàm là rất nhỏ. Nếu các bạn thử vài trường hợp khác nữa của `N, C, d`, chúng ta sẽ thấy sự sai khác vẫn là nhỏ. Điều này chứng tỏ đạo hàm chúng ta tính được coi là chính xác. (Vẫn có thể có bug, chỉ khi nào kết quả cuối cùng với dữ liệu thật là chấp nhận được thì ta mới có thể bỏ cụm từ 'có thể coi' đi).

Chú ý rằng, nếu `N, C, d` quá lớn, việc tính toán `numerical_grad` trở nên cực kỳ tốn thời gian và bộ nhớ. Chúng ta chỉ nên kiểm tra với những dữ liệu nhỏ.

13.4.4 Hàm chính cho training Softmax Regression

Sau khi đã có những hàm cần thiết và gradient được tính đúng, chúng ta có thể viết hàm chính có training Softmax Regression (theo SGD) như sau:

```
def softmax_regression(X, y, W_init, eta, tol = 1e-4, max_count = 10000):
    W = [W_init]
    C = W_init.shape[1]
    Y = convert_labels(y, C)
    it = 0
    N = X.shape[1]
    d = X.shape[0]

    count = 0
    check_w_after = 20
    while count < max_count:
        # mix data
        mix_id = np.random.permutation(N)
        for i in mix_id:
            xi = X[:, i].reshape(d, 1)
            yi = Y[:, i].reshape(C, 1)
            ai = softmax(np.dot(W[-1].T, xi))
            W_new = W[-1] + eta*xi.dot((yi - ai).T)
            count += 1
            # stopping criteria
            if count%check_w_after == 0:
                if np.linalg.norm(W_new - W[-check_w_after]) < tol:
                    return W
            W.append(W_new)
    return W
eta = .05
d = X.shape[0]
W_init = np.random.randn(d, C)

W = softmax_regression(X, y, W_init, eta)
# W[-1] is the solution, W is all history of weights
```

13.4.5 Hàm dự đoán class cho dữ liệu mới

Sau khi train Softmax Regression và tính được ma trận hệ số `W`, class của một dữ liệu mới có thể tìm được bằng cách xác định vị trí của giá trị lớn nhất ở đầu ra dự đoán (tương ứng với xác suất điểm dữ liệu rơi vào class đó là lớn nhất). Chú ý rằng, các class được đánh số là `0, 1, 2, ..., C`.

```
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 class are indexed by [0, 1, 2, ...., C-1]
    """
    A = softmax_stable(W.T.dot(X))
    return np.argmax(A, axis = 0)
```

13.5 Ví dụ với Python

13.5.1 Simulated data

Để minh họa cách áp dụng Softmax Regression, tôi tiếp tục làm trên *simulated data*.

Tạo ba cụm dữ liệu

```
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)

# each column is a datapoint
X = np.concatenate((X0, X1, X2), axis = 0).T
# extended data
X = np.concatenate((np.ones((1, 3*N)), X), axis = 0)
C = 3

original_label = np.asarray([0]*N + [1]*N + [2]*N).T
```

Phân bố của các dữ liệu được cho như hình dưới:

Thực hiện Softmax Regression

```
W_init = np.random.randn(X.shape[0], C)
W = softmax_regression(X, original_label, W_init, eta)
print(W[-1])
```

[[8.45809734 -3.88415491 -3.44660294] [-1.11205751 1.50441603 -0.76358758] [0.24484886 0.26085383 3.3658872]]

Kết quả thu được

Ta thấy rằng Softmax Regression đã tạo ra các vùng cho mỗi class. Kết quả này là chấp nhận được. Từ hình trên ta cũng thấy rằng *đường ranh giới* giữa các classes là đường thẳng. Tôi sẽ chứng minh điều này ở phần sau.

13.5.2 Softmax Regression cho MNIST

Các ví dụ trên đây được trình bày để giúp bạn đọc hiểu rõ Softmax Regression hoạt động như thế nào. Khi làm việc với các bài toán thực tế, chúng ta nên sử dụng các thư viện có sẵn, trừ khi bạn có thêm bớt vài số hạng nữa trong hàm mất mat.

Softmax Regression cũng được tích hợp trong hàm sklearn.linear_model.LogisticRegression của thư viện sklearn.

Để sử dụng Softmax Regression, ta cần thêm một vài thuộc tính nữa:

```
linear_model.LogisticRegression(C=1e5, solver = 'lbfgs', multi_class = 'multinomial')
```

Với Logistic Regression, `multi_class = 'ovr'` là giá trị mặc định, tương ứng với **one-vs-rest**. `solver = 'lbfgs'` là một phương pháp tối ưu cũng dựa trên gradient nhưng hiệu quả hơn và phức tạp hơn Gradient Descent. Bạn đọc có thể đọc thêm ở đây.

```
# %reset
import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.metrics import accuracy_score

mntrain = MNIST('../MNIST/')
mntrain.load_training()
Xtrain = np.asarray(mntrain.train_images)/255.0
ytrain = np.array(mntrain.train_labels.tolist())

mntest = MNIST('../MNIST/')
mntest.load_testing()
Xtest = np.asarray(mntest.test_images)/255.0
ytest = np.array(mntest.test_labels.tolist())

# train
logreg = linear_model.LogisticRegression(C=1e5,
        solver = 'lbfgs', multi_class = 'multinomial')
logreg.fit(Xtrain, ytrain)

# test
y_pred = logreg.predict(Xtest)
print "Accuracy: %.2f %%" %(100*accuracy_score(ytest, y_pred.tolist()))
```

Accuracy: 92.59

So với kết quả hơn 91

13.6 Thảo luận

13.6.1 Boundary tạo bởi Softmax Regression là linear

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 $a_j \geq a_k \Leftrightarrow z_j \geq z_k$, hay nói cách khác:

$$\mathbf{w}_j^T\mathbf{x} \geq \mathbf{w}_k^T\mathbf{x} \ \Leftrightarrow (\mathbf{w}_j - \mathbf{w}_k)^T\mathbf{x} \geq 0$$

Đây chính là một biểu thức tuyến tính. Vậy boundary tạo bởi Softmax Regression có dạng tuyến tính. (Xem thêm boundary tạo bởi Logistic Regression)

13.6.2 Softmax Regression là một trong hai classifiers phổ biến nhất

Softmax Regression cùng với Support Vector Machine (tôi sẽ trình bày sau vài bài nữa) là hai classifier phổ biến nhất được dùng hiện nay. Softmax Regression đặc biệt được sử dụng nhiều trong các mạng Neural có nhiều lớp (Deep Neural Networks hay DNN). Những lớp phía trước có thể được coi như một bộ Feature Extractor, lớp cuối cùng của DNN cho bài toán classification thường là Softmax Regression.

13.6.3 Source code

Các bạn có thể tìm thấy source code trong jupyter notebook này.

13.7 Tài liệu tham khảo

[1] Softmax Regression

[2] sklearn.linear_model.LogisticRegression

[3] Softmax function - Wikipedia

[4] Improving the way neural networks learn

14

Multi-layer Perceptron và Backpropagation

Vì bài này sử dụng khá nhiều công thức toán, bạn đọc được khuyến khích đọc Lưu ý về ký hiệu toán học.

14.1 Giới thiệu

Bài toán Supervised Learning, nói một cách ngắn gọn, là việc đi tìm một hàm số để với mỗi *input*, ta sử dụng hàm số đó để dự đoán *output*. Hàm số này được xây dựng dựa trên các cặp dữ liệu $(\mathbf{x}_i, \mathbf{y}_i)$ trong *training set*. Nếu *đầu ra dự đoán* (predicted output) gần với *đầu ra thực sự* (ground truth) thì đó được gọi là một thuật toán tốt (nhưng khi *đầu ra dự đoán quá giống với đầu ra thực sự* thì không hẳn đã tốt, tôi sẽ đề cập kỹ về hiện tượng trong bài tiếp theo).

14.1.1 PLA cho các hàm logic cơ bản

Chúng ta cùng xét khả năng biểu diễn (representation) của Perceptron Learning Algorithm (PLA) cho các bài toán binary vô cùng đơn giản: biểu diễn các hàm số logic NOT, AND, OR, và XOR (output bằng 1 nếu và chỉ nếu hai input khác nhau). Để có thể sử dụng PLA (output là 1 hoặc -1), chúng ta sẽ thay các giá trị bằng 0 của output của các hàm này bởi -1. Trong hàng trên của Hình 1 dưới đây, các điểm hình vuông màu xanh là các điểm có label bằng 1, các điểm hình tròn màu đỏ là các điểm có label bằng -1. Hàng dưới của Hình 1 là các mô hình perceptron với các hệ số tương ứng.

Nhận thấy rằng với các bài toán OR, AND, và OR, dữ liệu là *linearly separable*, vì vậy ta có thể tìm được các hệ số cho perceptron giúp biểu diễn chính xác mỗi hàm số. Xem ví dụ 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, predicted output đều giống với ground truth. Bạn đọc có thể tự kiểm chứng các hệ số trong hình với hàm AND và OR.

14.1.2 Biểu diễn hàm XOR với Neural Network.

Với hàm XOR, vì dữ liệu không *linearly separable*, tức không thể tìm được 1 đường thẳng giúp phân chia hai lớp xanh đỏ, nên bài toán vô nghiệm. Nếu thay PLA bằng Logistic Regression, tức thay hàm activation function từ *sgn* sang *sigmoid*, ta cũng không tìm được các hệ số thỏa mãn, vì về bản chất, Logistic Regression cũng chỉ tạo ra các đường biên 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.

Nhận thấy rằng nếu cho phép sử dụng hai đường thẳng, bài toán biểu diễn hàm XOR sẽ được giải quyết như Hình 2 (trái) dưới đây: Các hệ số tương ứng với hai đường thẳng trong Hình 2 (trái) được minh họa trên Hình 2 (phải) tại các node màu lục và lam (xin tạm gọi màu *cyan* này là *lam*. Tôi muốn sử dụng màu *lam* thật nhưng khi convert từ .pdf sang .png, nó lại giống màu *tím*). Đầu ra a_1 bằng 1 với các điểm nằm về phía (+) của đường thẳng $-2x_1 - 2x_2 + 3 = 0$, 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 $2x_1 + 2x_2 - 1 = 0$. Như vậy, hai đường thẳng này tạo ra hai *đầu ra* tại các node a_1, a_2. Vì hàm XOR chỉ có một đầu ra nên ta cần làm thêm một bước nữa: coi a_1, a_2 như là input của một PLA khác. Trong PLA mới này, input là các node màu lam (đừng quên node bias có giá trị bằng 1), output là các node màu đỏ. Các hệ số được cho trên Hình 2 (phải). Kiểm tra lại một chút, với các điểm hình vuông xanh (hình trái), $a_1^{(1)} = a_2^{(1)} = 1$, khi đó $a^{(2)} = \text{sgn}(1 + 1 - 1) = 1$. Với các điểm hình tròn đỏ, $a_1^{(1)} = -a_2^{(1)}$, vậy nên $a^{(2)} = \text{sgn}(a_1^{(1)} + a_2^{(1)} - 1) = \text{sgn}(-1) = -1$. Trong cả hai trường hợp, predicted ouput đều giống với ground truth. Vậy, nếu ta sử dụng 3 PLA tương ứng với các output $a_1^{(1)}, a_2^{(1)}, a^{(2)}$, ta sẽ biểu diễn được hàm XOR.

Ba PLA kể trên được xếp vào hai *layers*. Layer thứ nhất: input - lục, output - lam. Layer thứ hai: input - lam, output - đỏ. Ở đây, output của layer thứ nhất chính là input của layer thứ hai. Tổng hợp lại ta được một mô hình mà ngoài layer input (lục) và output (đỏ), ta còn có một layer nữa (lam). Mô hình này có tên gọi là Multi-layer Perceptron (MLP). Layer trung gian ở giữa còn được gọi là *hidden layer*.

Một vài lưu ý:

- Perceptron Learing Algorithm là một trường hợp của *single-layer neural network* với *activation fucntion* là hàm *sgn*. Trong khi đó, Perceptron là tên chung để chỉ các Neural Network với chỉ một input layer và một output tại output layer, không có hidden layer.

- Các *activation function* có thể là các nonlinear function khác, ví dụ như *sigmoid function* hoặc *tanh function*. Các *activation function* phải là nonlinear (phi tuyến), vì nếu không, nhiều layer hay một layer cũng là như nhau. Ví dụ với hai layer trong Hình 2, nếu *activation function* là một hàm linear (giả sử hàm $f(s) = s$), thì cả hai layer có thể được thay bằng một layer với ma trận hệ số $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ (tạm bỏ qua biases).

- Để cho đơn giản, tôi đã sử dụng ký hiệu $\mathbf{W}^{(l)T}$ để thay cho $(\mathbf{W}^{(l)})^T$ (ma trận chuyển vị). Trong Hình 2 (phải), tôi sử dụng ký hiệu ma trận $\mathbf{W}^{(2)}$, mặc dù đúng ra nó phải là

vector, để biểu diễn tổng quát cho trường hợp output layer có thể có nhiều hơn 1 node. Tương tự với bias $\mathbf{b}^{(2)}$.

- Khác với các bài trước về Neural Networks, khi làm việc với MLP, ta nên tách riêng phần biases và ma trận hệ số ra. Điều này đồng nghĩa với việc vector input $\mathbf{x}$ là vector KHÔNG mở rộng.

14.2 Các ký hiệu và khái niệm

14.2.1 Layers

Ngoài *Input layers* và *Output layers*, một Multi-layer Perceptron (MLP) có thể có nhiều *Hidden layers* ở giữa. Các *Hidden layers* theo thứ tự từ input layer đến output layer được đánh số thứ thự là *Hidden layer 1*, *Hidden layer 2*, ... Hình 3 dưới đây là một ví dụ với 2 Hidden layers.

Số lượng layer trong một MLP được tính bằng số hidden layers cộng với 1. Tức là khi đếm số layers của một MLP, ta không tính input layers. Số lượng layer trong một MLP thường được ký hiệu là L. Trong Hình 3 trên đây, $L = 3$.

14.2.2 Units

Một *node* hình tròn trong một layer được gọi là một unit. Unit ở các input layer, hidden layers, và output layer được lần lượt gọi là input unit, hidden unit, và output unit. Đầu vào của các hidden layer được ký hiệu bởi z, đầu ra của mỗi unit thường được ký hiệu là a (thể hiện *activation*, tức giá trị của mỗi unit sau khi ta áp dụng activation function lên z). 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 $\mathbf{a}^{(l)} \in \mathbb{R}^{d^{(l)}}$.

Khi làm việc với những Neural Networks phức tạp, cách tốt nhất để hạn chế lỗi là viết cụ thể chiều của mỗi ma trận hay vector ra, bạn sẽ thấy rõ hơn trong phần sau.

14.2.3 Weights và Biases

Có L ma trận trọng số cho một MLP có L layers. Các ma trận này được ký hiệu là $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}, l = 1, 2, \ldots, L$ 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 từ j của layer thứ (l). Các biases 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 4. Khi tối ưu một MLP cho một công việc nào đó, chúng ta cần đi tìm các weghts và biases này.

Tập hợp các weights và biases lần lượt được ký hiệu là $\mathbf{W}$ và $\mathbf{b}$.

14.2.4 Activation functions

(Phần này chú yếu được dịch lại từ: http://cs231n.github.io/neural-networks-1/)

Mỗi output của một unit (trừ các input units) được tính dựa vào công thức:

$$a_i^{(l)} = f(\mathbf{w}_i^{(l)T}\mathbf{a}^{(l-1)} + b_i^{(l)})$$

Trong đó $f(.)$ là một (nonlinear) activation function. Ở dạng vector, biểu thức bên trên được viết là:

$$\mathbf{a}^{(l)} = f(\mathbf{W}^{(l)T}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)})$$

Khi activation function $f(.)$ được áp dụng cho một ma trận (hoặc vector), ta hiểu rằng nó được áp dụng cho *từng thành phần của ma trận đó*. Sau đó các thành phần này được sắp xếp lại đúng theo thứ tự để được một ma trận có kích thước bằng với ma trận input. Trong tiếng Anh, việc áp dụng lên từng phần tử như thế này được gọi là *element-wise*.

Hàm *sgn* không được sử dụng trong MLP

Hàm *sgn* (còn gọi là *hard-threshold*) chỉ được sử dụng trong PLA, mang mục đích giáo dục nhiều hơn. Trong thực tế, hàm *sgn* không được sử dụng vì hai lý do: đầu ra là *discrete*, và đạo hàm tại hầu hết các điểm bằng 0 (trừ điểm 0 không có đạo hàm). Việc đạo hàm bằng 0 này khiến cho các thuật toán gradient-based (ví dụ như Gradient Descent) không hoạt động!

Sigmoid và tanh

Hàm *sigmoid* có dạng $f(s) = 1/(1 + \exp(-s))$ với đồ thị như trong Hình 5 (trái). 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. Hàm số này được sử dụng nhiều trong quá khứ ví có đạo hàm rất *đẹp*. Những năm gần đây, hàm số này ít khi được sử dụng. Nó có một nhược điểm cơ bản:

- *Sigmoid saturate and kill gradients*: Một nhược điểm dễ nhận thấy là khi đầu vào có trị tuyệt đối lớn (rất âm hoặc rất dương), gradient của hàm số này 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. Bạn đọc sẽ hiểu rõ hơn phần này trong phần Backpropagation.

Hàm *tanh* cũng có nhược điểm tương tự về việc gradient rất nhỏ với các đầu vào có trị tuyệt đối lớn.

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 5 (trái)). Nó có công thức toán học $f(s) = \max(0, s)$ - rất đơn giản. Ưu điểm chính của nó là:

- ReLU được chứng minh giúp cho việc training các *Deep Networks* nhanh hơn rất nhiều (theo Krizhevsky et al.). Hình 5 (phải) so sánh sự hội tụ của SGD khi sử dụng hai activation function khác nhau: 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 với gradient bằng 1 nếu đầu vào lớn hơn 0, bằng 0 nếu đầu vào nhỏ hơn 0.
- Mặc dù hàm ReLU không có đạo hàm tại $s = 0$, trong thực nghiệm, người ta vẫn thường định nghĩa $\text{ReLU}'(0) = 0$ và khẳng định thêm rằng, xác suất để input của một unit bằng 0 là rất nhỏ.

Hàm ReLU có nhiều biến thể khác như Noisy ReLU, Leaky ReLu, ELUs). Tôi xin phép dừng phần này ở đây vì chưa có ý định đi sâu vào Deep Neural Networks.

Một vài lưu ý

- Output layer nhiều khi không có activation function mà sử dụng trực tiếp giá trị đầu vào $z_i^{(l)}$ của mỗi unit. Hoặc nói một cách khác, activation function chính là hàm *identity*, tức đầu ra bằng đầu vào. Với các bài toán classification, output layer thường là một Softmax Regression layer giúp tính xác suất để một điểm dữ liệu rơi vào mỗi class.
- Mặc dù activation function cho mỗi unit có thể khác nhau, trong cùng một network, activation như nhau thường được sử dụng. Điều này giúp cho việc tính toán được đơn giản hơn.

14.3 Backpropagation

Phần này khá nặng về Đại Số Tuyến Tính, bạn đọc không muốn hiểu backpropagation có thể bỏ qua để đọc tiếp phần Ví dụ với Python.

Phương pháp phổ biến nhất để tối ưu MLP vẫn là Gradient Descent (GD). Để áp dụng GD, chúng ta cần tính được gradient của hàm mất mát theo từng ma trận trọng số $\mathbf{W}^{(l)}$ và vector bias $\mathbf{b}^{(l)}$. Trước hết, chúng ta cần tính *predicted output* $\hat{\mathbf{y}}$ với một input $\mathbf{x}$:

$$\mathbf{a}^{(0)} = \mathbf{x} \tag{14.1}$$
$$z_i^{(l)} = \mathbf{w}_i^{(l)T}\mathbf{a}^{(l-1)} + b_i^{(l)} \tag{14.2}$$
$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, \quad l = 1, 2, \ldots, L \tag{14.3}$$
$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}), \quad l = 1, 2, \ldots, L \tag{14.4}$$
$$\hat{\mathbf{y}} = \mathbf{a}^{(L)} \tag{14.5}$$

Bước này được gói là *feedforward* vì cách tính toán được thực hiện từ đầu đến cuối của network. MLP cũng được gọi

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 layers và biases 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 gradient-based methods (mà Gradient Descent là một ví dụ), chúng ta cần tính được:

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}}; \frac{\partial J}{\partial \mathbf{b}^{(l)}}, \quad l = 1, 2, \ldots, L$$

Một ví dụ của hàm mất mát là hàm Mean Square Error (MSE) tức *trung bình của bình phương lỗi*.

$$J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y}) = \frac{1}{N}\sum_{n=1}^{N} \|\|\mathbf{y}_n - \hat{\mathbf{y}}_n\|\|_2^2 \tag{14.6}$$
$$= \frac{1}{N}\sum_{n=1}^{N} \|\|\mathbf{y}_n - \mathbf{a}_n^{(L)}\|\|_2^2 \tag{14.7}$$

Với N là số cặp dữ liệu $(\mathbf{x}, \mathbf{y})$ trong tập training.

Theo những công thức ở trên, việc tính toán trực tiếp giá trị này là cực kỳ phức tạp vì hàm mất mát không phụ thuộc trực tiếp vào các hệ số. Phương pháp phổ biến nhất được dùng có tên là Backpropagation giúp tính gradient ngược từ layer cuối cùng đến layer đầu tiên. Layer cuối cùng được tính toán trước vì nó *gần gũi* hơn với *predicted outputs* và hàm mất mát. Việc tính toán gradient của các layer trước được thực hiện dựa trên một quy tắc quen thuộc có tên là *chain rule*, tức *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 lớp cuối cùng:

$$\frac{\partial J}{\partial w_{ij}^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} . \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} \tag{14.8}$$

$$= e_j^{(L)} a_i^{(L-1)} \tag{14.9}$$

Trong đó $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$ thường là một đạilượng *dễ 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 như thế, đạo hàm của hàm mất mát theo bias của layer cuối cùng là:

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} . \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = e_j^{(L)}$$

Với đạo hàm theo hệ số ở các lớp l *thấp hơn*, chúng ta hay xem hình dưới đây. Ở đây, tại mỗi unit, tôi đã viết riêng đầu vào z và đầu ra a để các bạn tiện theo dõi.

Dựa vào hính trên, ta có thể tính được:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} . \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{14.10}$$

$$= e_j^{(l)} a_i^{(l-1)} \tag{14.11}$$

với:

$$e_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} . \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \tag{14.12}$$

$$= \left(\sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} . \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f'(z_j^{(l)}) \tag{14.13}$$

$$= \left(\sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f'(z_j^{(l)}) \tag{14.14}$$

$$= \left(\mathbf{w}_{j:}^{(l+1)} \mathbf{e}^{(l+1)} \right) f'(z_j^{(l)}) \tag{14.15}$$

$$\tag{14.16}$$

trong đó $\mathbf{e}^{(l+1)} = [e_1^{(l+1)}, e_2^{(l+1)}, ..., e_{d^{(l+1)}}^{(l+1)}]^T \in \mathbb{R}^{d^{(l+1)} \times 1}$ 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, tôi mặc định ký hiệu nó cho vector *cột*).

Dấu sigma tính tổng ở hàng thứ hai trong phép tính trên xuất hiện vì $a_j^{(l)}$ *đóng góp* vào việc tính tất cả các $z_k^{(l+1)}, k = 1, 2, \ldots, d^{(l+1)}$. Biểu thức đạo hàm ngoài dấu ngoặc lớn là vì $a_j^{(l)} = f(z_j^{(l)})$. 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:

$$\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)}$$

Nhận thấy rằng trong các công thức trên đây, việc tính các $e_j^{(k)}$ đóng một vài trò quan trọng. Hơn nữa, để tính được giá trị này, ta cần tính được các $e_k^{(l+1)}$. 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:

14.3.1 Backpropagation cho Stochastic Gradient Descent

Đạo hàm theo từng hệ số $w_{ij}^{(l)}, b_i^{(l)}$

<hr> 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 *activation* $\mathbf{a}^{(l)}$ tại mỗi layer. 2. Với mỗi unit j ở output layer, tính

$$e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$$

3. Từ đó suy ra:

$$\frac{\partial J}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} e_j^{(L)} \tag{14.17}$$

$$\frac{\partial J}{\partial b_j^{(L)}} = e_j^{(L)} \tag{14.18}$$

4. Với $l = L-1, L-2, ..., 1$, tính:

$$e_j^{(l)} = \left(\mathbf{w}_{j:}^{(l+1)} \mathbf{e}^{(l+1)}\right) f'(z_j^{(l)})$$

5. Cập nhật đạo hàm cho từng hệ số:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = a_i^{(l-1)} e_j^{(l)} \tag{14.19}$$

$$\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)} \tag{14.20}$$

<hr>

Đạo hàm theo ma trận $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$

Việc tính toán theo từng hệ số như trên chỉ phù hợp cho việc hiểu nguyên lý tính toán, trong khi lập trình, ta cần tìm cách thu gọn chúng về dạng vector và ma trận để tăng tốc độ cho thuật toán. Đặt $\mathbf{e}^{(l)} = [e_1^{(l)}, e_2^{(l)}, ..., e_{d^{(l)}}^{(l)}]^T \in \mathbb{R}^{d^{(l)} \times 1}$. Ta sẽ có quy tắc tính như sau:

<hr> 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 *activation* $\mathbf{a}^{(l)}$ tại mỗi layer. 2. Với output layer, tính:

$$\mathbf{e}^{(L)} = \frac{\partial J}{\partial \mathbf{z}^{(L)}}$$

3. Từ đó suy ra:

$$\frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{a}^{(L-1)}\mathbf{e}^{(L)T} \tag{14.21}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(L)}} = \mathbf{e}^{(L)} \tag{14.22}$$

4. Với $l = L-1, L-2, ..., 1$, tính:

$$\mathbf{e}^{(l)} = \left(\mathbf{W}^{(l+1)}\mathbf{e}^{(l+1)}\right) \odot f'(\mathbf{z}^{(l)})$$

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. 5. Cập nhật đạo hàm cho ma trận trọng số và vector biases:

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \mathbf{a}^{(l-1)}\mathbf{e}^{(l)T} \tag{14.23}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \mathbf{e}^{(l)} \tag{14.24}$$

<hr>

Chú ý: Biểu thức tính đạo hàm trong dòng trên của bước 3 có thể khiến bạn đặt câu hỏi: tại sao lại là $\mathbf{a}^{(L-1)}\mathbf{e}^{(L)T}$ mà không phải là $\mathbf{a}^{(L-1)T}\mathbf{e}^{(L)}$, $\mathbf{e}^{(L)T}\mathbf{a}^{(L-1)}$, hay $\mathbf{e}^{(L)}\mathbf{a}^{(L-1)T}$? *Quy tắc bỏ túi* cần nhớ là **chiều của hai ma trận ở hai vế phải như nhau**. Thử một chút, vế trái là đạo hàm theo $\mathbf{W}^{(L)}$ là một đại lượng có chiều (*dimension*, not *afternoon*) bằng chiều của ma trận này, tức chiều là $\mathbb{R}^{d^{(L-1)} \times d^{(L)}}$. Vế phải, $\mathbf{e}^{(L)} \in \mathbf{R}^{d^{(L)} \times 1}$, $\mathbf{a}^{(L-1)} \in \mathbb{R}^{d^{(L-1)} \times 1}$. Để hai vế có chiều bằng nhau thì ta phải lấy $\mathbf{a}^{(L-1)}\mathbf{e}^{(L)T}$. Cũng chú ý thêm **rằng đạo hàm theo một ma trận của một hàm số nhận giá trị thực (scalar) sẽ có chiều bằng với chiều của ma trận đó!!**

14.3.2 Backpropagation cho Batch (mini-batch) Gradient Descent

Nếu chúng ta muốn thực hiện Batch hoặc mini-batch Gradient Descent thì sao? Trong thực tế, mini-batch GD được sử dụng nhiều nhất. 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 thế, $\mathbf{E}^{(l)} \in \mathbb{R}^{d^{(l)} \times N}$. Và ta cũng có thể suy ra công thức cập nhật như sau.

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:

$$\mathbf{E}^{(L)} = \frac{\partial J}{\partial \mathbf{Z}^{(L)}}$$

3. Từ đó suy ra:

$$\frac{\partial J}{\partial \mathbf{W}^{(L)}} = \mathbf{A}^{(L-1)} \mathbf{E}^{(L)T} \tag{14.25}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(L)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(L)} \tag{14.26}$$

$$\mathbf{E}^{(l)} = \left(\mathbf{W}^{(l+1)} \mathbf{E}^{(l+1)}\right) \odot f'(\mathbf{Z}^{(l)}) \tag{14.27}$$

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. 5. Cập nhật đạo hàm cho ma trận trọng số và vector biases:

$$\frac{\partial J}{\partial \mathbf{W}^{(l)}} = \mathbf{A}^{(l-1)} \mathbf{E}^{(l)T} \tag{14.28}$$

$$\frac{\partial J}{\partial \mathbf{b}^{(l)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(l)} \tag{14.29}$$

<hr>

Mặc dù khi làm thực nghiệm, các công cụ có hỗ trợ việc tự động tính Backpropagation, tôi vẫn không muốn bỏ qua phần này. Hiểu backpropagation rất quan trọng! Xem thêm Yes you should understand backprop.

14.4 Ví dụ trên Python

Source code cho ví dụ này có thể được xem tại đây.

Ví dụ tôi nêu trong mục này mang mục đích giúp các bạn hiểu thực sự cách lập trình cho backpropagation. Khi làm thực nghiệm, chúng ta sử dụng các thư viện sẵn có giúp tính backpropagation. Ví dụ như Sklearn cho MLP.

Để kiểm chứng lại những gì tôi viết trên đây có đúng không, chúng ta cùng xem một ví dụ. Ý tưởng trong ví dụ này được lấy từ CS231n Convolutional Neural Networks for Visual Recognition, phần code dưới đây tôi viết lại cho phù hợp với những tính toán và ký hiệu phía trên.

14.4.1 Tạo dữ liệu giả

Trước hết, ta tạo dữ liệu cho 3 classes mà không có hai class nào là *linearly separable*:

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import math
import numpy as np
import matplotlib.pyplot as plt

N = 100 # number of points per class
d0 = 2 # dimensionality
C = 3 # number of classes
X = np.zeros((d0, N*C)) # data matrix (each row = single example)
y = np.zeros(N*C, dtype='uint8') # class labels

for j in xrange(C):
  ix = range(N*j,N*(j+1))
  r = np.linspace(0.0,1,N) # radius
  t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
  X[:,ix] = np.c_[r*np.sin(t), r*np.cos(t)].T
  y[ix] = j
# lets visualize the data:
# plt.scatter(X[:N, 0], X[:N, 1], c=y[:N], s=40, cmap=plt.cm.Spectral)

plt.plot(X[0, :N], X[1, :N], 'bs', markersize = 7);
plt.plot(X[0, N:2*N], X[1, N:2*N], 'ro', markersize = 7);
plt.plot(X[0, 2*N:], X[1, 2*N:], 'g^', markersize = 7);
# plt.axis('off')
plt.xlim([-1.5, 1.5])
plt.ylim([-1.5, 1.5])
cur_axes = plt.gca()
cur_axes.axes.get_xaxis().set_ticks([])
cur_axes.axes.get_yaxis().set_ticks([])

plt.savefig('EX.png', bbox_inches='tight', dpi = 600)
plt.show()
```

Với dữ liệu được phân bố thế này, Softmax Regression không thể thực hiện được vì Bounray giữa các class tạo bởi Softmax Regression có dạng linear. Chúng ta hãy làm một thí nghiệm nhỏ bằng cách thêm một *Hidden layer* vào giữa Input layer vả output layer của Softmax Regression. Activation function của Hidden layer là hàm ReLU: $f(s) = \max(s, 0)$, $f'(s) = 0$ if $s \leq 0$, $f'(s) = 1$ otherwise.

Bây giờ chúng ta sẽ áp dụng Batch Gradient Descent cho bài toán này (vì lượng dữ liệu là nhỏ). Trước hết cần thực tìm công thức tính các activation và output.

14.4.2 Tính toán Feedforward

$$\mathbf{Z}^{(1)} = \mathbf{W}^{(1)T}\mathbf{X} \tag{14.30}$$
$$\mathbf{A}^{(1)} = \max(\mathbf{Z}^{(1)}, \mathbf{0}) \tag{14.31}$$
$$\mathbf{Z}^{(2)} = \mathbf{W}^{(2)T}\mathbf{A}^{(1)} \tag{14.32}$$
$$\hat{\mathbf{Y}} = \mathbf{A}^{(2)} = \text{softmax}(\mathbf{Z}^{(2)}) \tag{14.33}$$

Hàm mất mát được tính như sau:

$$J \triangleq J(\mathbf{W}, \mathbf{b}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ji}\log(\hat{y}_{ji})$$

Ở đây, tôi đã cho thêm thừa số $\frac{1}{N}$ để tránh hiện tượng tổng quá lớn với Batch GD. Về mặt toán học, thừa số này không làm thay đổi nghiệm của bài toán.

14.4.3 Tính toán Backpropagation

Áp dụng quy tắc như đã trình bày ở trên và

$$\mathbf{E}^{(2)} = \frac{\partial J}{\partial \mathbf{Z}^{(2)}} = \frac{1}{N}(\hat{\mathbf{Y}} - \mathbf{Y}) \tag{14.34}$$
$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \mathbf{A}^{(1)}\mathbf{E}^{(2)T} \tag{14.35}$$
$$\frac{\partial J}{\partial \mathbf{b}^{(2)}} = \sum_{n=1}^{N}\mathbf{e}_n^{(2)} \tag{14.36}$$
$$\mathbf{E}^{(1)} = \left(\mathbf{W}^{(2)}\mathbf{E}^{(2)}\right) \odot f'(\mathbf{Z}^{(1)}) \tag{14.37}$$
$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \mathbf{A}^{(0)}\mathbf{E}^{(1)T} = \mathbf{X}\mathbf{E}^{(1)T} \tag{14.38}$$
$$\frac{\partial J}{\partial \mathbf{b}^{(1)}} = \sum_{n=1}^{N}\mathbf{e}_n^{(1)} \tag{14.39}$$
$$\tag{14.40}$$

Từ đó ta có thể bắt đầu lập trình như sau:

14.4.4 Một số hàm phụ trợ

```
def softmax(V):
    e_V = np.exp(V - np.max(V, axis = 0, keepdims = True))
    Z = e_V / e_V.sum(axis = 0)
    return Z

## One-hot coding
from scipy import sparse
def convert_labels(y, C = 3):
    Y = sparse.coo_matrix((np.ones_like(y),
        (y, np.arange(len(y)))), shape = (C, len(y))).toarray()
    return Y

# cost or loss function
def cost(Y, Yhat):
    return -np.sum(Y*np.log(Yhat))/Y.shape[1]
```

14.4.5 Phần chương trình chính

```
d0 = 2
d1 = h = 100 # size of hidden layer
d2 = C = 3
# initialize parameters randomly
W1 = 0.01*np.random.randn(d0, d1)
b1 = np.zeros((d1, 1))
W2 = 0.01*np.random.randn(d1, d2)
b2 = np.zeros((d2, 1))

Y = convert_labels(y, C)
N = X.shape[1]
eta = 1 # learning rate
for i in xrange(10000):
    ## Feedforward
    Z1 = np.dot(W1.T, X) + b1
    A1 = np.maximum(Z1, 0)
    Z2 = np.dot(W2.T, A1) + b2
    Yhat = softmax(Z2)

    # print loss after each 1000 iterations
    if i %1000 == 0:
        # compute the loss: average cross-entropy loss
        loss = cost(Y, Yhat)
        print("iter %d, loss: %f" %(i, loss))

    # backpropagation
    E2 = (Yhat - Y )/N
    dW2 = np.dot(A1, E2.T)
    db2 = np.sum(E2, axis = 1, keepdims = True)
    E1 = np.dot(W2, E2)
    E1[Z1 <= 0] = 0 # gradient of ReLU
    dW1 = np.dot(X, E1.T)
    db1 = np.sum(E1, axis = 1, keepdims = True)

    # Gradient Descent update
    W1 += -eta*dW1
    b1 += -eta*db1
```

```
    W2 += -eta*dW2
    b2 += -eta*db2
```

iter 0, loss: 1.098815 iter 1000, loss: 0.150974 iter 2000, loss: 0.057996 iter 3000, loss: 0.039621 iter 4000, loss: 0.032148 iter 5000, loss: 0.028054 iter 6000, loss: 0.025346 iter 7000, loss: 0.023311 iter 8000, loss: 0.021727 iter 9000, loss: 0.020585

14.4.6 Kết quả

Như vậy, cứ sau 1000 vòng lặp, hàm mất mát giảm dần. Bây giờ chúng ta cùng áp dụng ngược network này vào phân loại *dữ liệu training*:

```
Z1 = np.dot(W1.T, X) + b1
A1 = np.maximum(Z1, 0)
Z2 = np.dot(W2.T, A1) + b2
predicted_class = np.argmax(Z2, axis=0)
print('training accuracy: %.2f %%' % (100*np.mean(predicted_class == y)))
```

training accuracy: 99.33

Vậy là trong 300 điểm, chỉ có 2 điểm bị phân loại sai! Dưới đây là hình minh hoạ *khu vực* của mỗi class:

Hai điểm bị phân lớp sai có lẽ nằm gần khu vực trung tâm.

Vậy là chỉ thêm 1 hidden layer, Neural Network đã có thể xây dựng được boundary *phi tuyến*. Kết luận đầu tiên ở đây là khả năng biểu diễn của MLP tốt hơn rất nhiều so với 1-layer Neural Network.

Kết quả bên trên được thực hiện khi số lượng units trong hidden layer là `d1 = 100`. Chúng ta thử thay đổi giá trị này bởi `d1 = 5, 10, 15, 20` xem kết quả khác nhau như thế nào. Dưới đây là hình mình họa:

Có một vài nhận xét như sau:

- Khi số lượng hidden units tăng lên, độ chính xác của mô hình tạo được cũng tăng lên.
- Với `d1 = 5`, đường phân định giữa ba classes gần như là đường thẳng.
- Với `d1 = 15`, mặc dù kết quả đã đạt 99.33
- Với `d1 = 20`, kết quả nhận được đã tương đối giống với `d1 = 100`. Mặc dù các đường boundary không được trơn tru cho lắm.

14.5 Thảo luận

- Người ta đã chứng minh được rằng, với một hàm số liên tục bất kỳ $f(x)$ và một số $\varepsilon > 0$, luôn luôn tồn tại một Neural Network với predicted output có dạng $g(x)$ với một hidden layer (với số hidden units đủ lớn và *nonlinear* activation function phù hợp) sao cho với mọi $x, \|f(x) - g(x)\| < \varepsilon$. Nói một cách khác, Neural Network có khả năng xấp xỉ bất kỳ hàm liên tục nào.

- Trên thực tế, việc tìm ra số lượng hidden units và *nonlinear* activation function nói trên nhiều khi bất khả thi. Thay vào đó, thực nghiệm chứng minh rằng Neural Networks với nhiều hidden layers kết hợp với các *nonlinear* activation function (đơn giản như ReLU) có khả năng xấp xỉ (khả năng biểu diễn) training data tốt hơn.

- Khi số lượng hidden layers 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 khia 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 training data, nhưng lại không biểu diễn tốt test data. Hiện tượng này gọi là Overfitting, tôi sẽ trình bày trong bài sau.

- Nếu mọi units của một layer được kết nối với mọi unit của layer tiếp theo (như chúng ta đang xét trong bai này), ta gọi đó là fully connected layer (kết nối hoàn toàn). Neural Networks với toàn fully connected layer ít được sử dụng trong thực tế. Thay vào đó, có nhiều phương pháp giúp làm giảm độ phức tạp của mô hình bằng cách giảm số lượng kết nối bằng cách cho nhiều kết nối bằng 0 (ví dụ, sparse autoencoder), hoặc các hệ số được ràng buộc giống nhau (để giảm số hệ số cần tối ưu) (ví dụ, Convolutional Neural Networks (CNNs / ConvNets)). Bạn đọc muốn tìm hiểu thêm có thể bắt đầu tại đây.

- Đây là bài cuối cùng trong chuỗi bài về Neural Networks. Viết một bài về Deep Learning sẽ tốn thời gian hơn rất nhiều, trong 1 tuần tôi không đủ khả năng hoàn thành được. Bài tiếp theo tôi sẽ nói về Overfitting, sau đó chuyển sang một phương pháp classification rất phổ biến khác: Support Vector Machine.

- Về backpropagation, có rất nhiều điều phải nói nữa. Nếu có thể, tôi xin phép được trình bày sau. Bài này cũng đã đủ dài.

14.6 Tài liệu tham khảo

[1] Neural Networks Part 1: Setting up the Architecture - Andrej Karpathy

[2] Neural Networks, Case study - Andrej Karpathy

[3] Lecture Notes on Sparse Autoencoders - Andrew Ng

[4] Yes you should understand backprop

[5] Backpropagation, Intuitions - Andrej Karpathy

[6] How the backpropagation algorithm works - Michael Nielsen

15

Overfitting

Overfitting không phải là một thuật toán trong Machine Learning. Nó 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.

15.1 Giới thiệu

Đây là một câu chuyện của chính tôi khi lần đầu biết đến Machine Learning.

Năm thứ ba đại học, một thầy giáo có giới thiệu với lớp tôi về Neural Networks. Lần đầu tiên nghe thấy khái niệm này, chúng tôi hỏi thầy mục đích của nó là gì. Thầy nói, về cơ bản, từ dữ liệu cho trước, chúng ta cần tìm một hàm số để biến các các điểm đầu vào thành các điểm đầu ra tương ứng, không cần chính xác, chỉ cần xấp xỉ thôi.

Lúc đó, vốn là một học sinh chuyên toán, làm việc nhiều với đa thức ngày cấp ba, tôi đã quá tự tin trả lời ngay rằng Đa thức Nội suy Lagrange có thể làm được điều đó, miễn là các điểm đầu vào khác nhau đôi một! Thầy nói rằng "những gì ta biết chỉ là nhỏ xíu so với những gì ta chưa biết". Và đó là những gì tôi muốn bắt đầu trong bài viết này.

Nhắc lại một chút về Đa thức nội suy Lagrange: Với N cặp điểm dữ liệu $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$ với các x_i khác nhau đôi một, luôn tìm được một đa thức $P(.)$ bậc không vượt quá $N-1$ sao cho $P(x_i) = y_i,\ \forall i = 1, 2, \ldots, N$. Chẳng phải điều này giống với việc ta đi tìm một mô hình phù hợp (fit) với dữ liệu trong bài toán Supervised Learning hay sao? Thậm chí điều này còn tốt hơn vì trong Supervised Learning ta chỉ cần xấp xỉ thôi.

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*, là điều mà khi xây dựng mô hình, chúng ta luôn cần tránh. Để có cái nhìn đầu tiên về overfitting, chúng ta cùng xem Hình dưới đây. Có 50 điểm dữ liệu được tạo bằng một đa thức bậc ba cộng thêm nhiễu. Tập dữ liệu này được chia làm hai, 30 điểm dữ liệu màu đỏ cho training data, 20 điểm dữ liệu màu vàng cho test data. Đồ thị của đa thức bậc ba này được cho bởi đường màu xanh lục. Bài

toán của chúng ta 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 dữ liệu đã cho.

Với những gì chúng ta đã biết từ bài Linear Regression, với loại dữ liệu này, chúng ta có thể áp dụng Polynomial Regression. Bài toán này hoàn toàn có thể được giải quyết bằng Linear Regression với dữ liệu mở rộng cho một cặp điểm (x, y) là $(\mathbf{x}, y)$ với $\mathbf{x} = [1, x, x^2, x^3, \ldots, x^d]^T$ cho đa thức bậc d. Điều quan trọng là chúng ta cần tìm bậc d của đa thức cần tìm.

Rõ ràng là một đa thức bậc không vượt quá 29 có thể *fit* được hoàn toàn với 30 điểm trong training data. Chúng ta cùng xét vài giá trị $d = 2, 4, 8, 16$. Với $d = 2$, mô hình không thực sự tốt vì mô hình dự đoán quá khác so với mô hình thực. Trong trường hợp này, ta nói mô hình bị *underfitting*. Với $d = 8$, với các điểm dữ liệu trong khoảng của training data, 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* dữ liệu trong khoảng đang xét, và *quá fit*, tức *không được mượt* trong khoảng dữ liệu training. Việc *quá fit* trong trường hợp bậc 16 không tốt vì mô hình đang cố gắng mô tả *nhiễu* hơn là dữ liệu. Hai trường hợp đa thức bậc cao này được gọi là *Overfitting*.

Nếu bạn nào biết về Đa thức nội suy Lagrange thì có thể hiểu được hiện tượng sai số lớn với các điểm nằm ngoài khoảng của các điểm đã cho. Đó chính là lý do phương pháp đó có từ "nội suy", với các trường hợp "ngoại suy", kết quả thường không chính xác.

Với $d = 4$, ta được mô hình dự đoán khá giống với mô hình thực. Hệ số bậc cao nhất tìm được rất gần với 0 (xem kết quả trong source code), vì vậy đa thực bậc 4 này khá gần với đa thức bậc 3 ban đầu. Đây chính là một mô hình tốt.

Overfitting là hiện tượng mô hình tìm được *quá khớp* với dữ liệu training. Việc *quá khớp* này có thể dẫn đến việc dự đoán nhầm nhiễu, và chất lượng mô hình không còn tốt trên dữ liệu test nữa. Dữ liệu test được giả sử là không được biết trước, và không được sử dụng để xây dựng các mô hình Machine Learning.

Về cơ bản, overfitting xảy ra khi mô hình quá phức tạp để mô phỏng training data. Điều này đặc biệt xảy ra khi lượng dữ liệu training quá nhỏ trong khi 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. Trong Multi-layer Perceptron, độ phức tạp của mô hình có thể được coi là số lượng hidden layers và số lượng units trong các hidden layers.

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 data và test data. Dưới đây là hai đại lượng đơn giản, với giả sử $\mathbf{y}$ là đầu ra thực sự (có thể là vector), và $\hat{\mathbf{x}}$ là đầu ra dự đoán bởi mô hình:

Train error: Thường là 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 Regression, đại lượng này thường được định nghĩa:

$$\text{train error} = \frac{1}{N_{\text{train}}} \sum_{\text{training set}} \|\mathbf{y} - \hat{\mathbf{y}}\|_p^2$$

với p thường bằng 1 hoặc 2.

Với Classification, trung bình cộng của cross entropy có thể được sử dụng.

Test error: Tương tự như trên nhưng áp dụng mô hình tìm được vào **test data**. Chú ý rằng, khi xây dựng mô hình, ta không được sử dụng thông tin trong tập dữ liệu test. Dữ liệu test chỉ được dùng để đánh giá mô hình. Với Regression, đại lượng này thường được định nghĩa:

$$\text{test error} = \frac{1}{N_{\text{test}}} \sum_{\text{test set}} \|\mathbf{y} - \hat{\mathbf{y}}\|_p^2$$

với p giống như p trong cách tính *train error* phía trên.

Việc lấy trung bình là quan trọng vì lượng dữ liệu trong hai tập hợp training và test có thể chênh lệch rất nhiều.

Một mô hình được coi là tốt (fit) nếu cả *train error* và *test error* đều thấp. Nếu *train error* thấp nhưng *test error* cao, ta nói mô hình bị overfitting. Nếu *train error* cao và *test error* cao, ta nói mô hình bị underfitting. Nếu *train error* cao nhưng *test error* thấp, tôi không biết tên của mô hình này, vì cực kỳ may mắn thì hiện tượng này mới xảy ra, hoặc có chỉ khi tập dữ liệu test quá nhỏ.

Chúng ta cùng đi vào phương pháp đầu tiên

15.2 Validation

15.2.1 Validation

Chúng ta vẫn quen với việc chia tập dữ liệu ra thành hai tập nhỏ: training data và test data. Và một điều tôi vẫn muốn nhắc lại là khi xây dựng mô hình, ta không được sử dụng test data. Vậy làm cách nào để biết được chất lượng của mô hình với *unseen data* (tức dữ liệu chưa nhìn thấy bao giờ)?

Phương pháp đơn giản nhất là *trích* từ tập training data 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 là phần còn lại của training set ban đầu**. Train error được tính trên training set mới này, và có một khái niệm nữa được định nghĩa tương tự như trên *validation error*, tức error được tính trên tập validation.

Với khái niệm mới này, ta tìm mô hình sao cho cả *train eror* và *validation error* đều nhỏ, qua đó có thể dự đoán được rằng *test error* cũng nhỏ. Phương pháp thường được sử dụng là sử dụng nhiều mô hình khác nhau. Mô hình nào cho *validation error* nhỏ nhất sẽ là mô hình tốt.

Thông thường, ta bắt đầu từ mô hình đơn giản, sau đó tăng dần độ phức tạp của mô hình. Tới khi nào *validation error* có chiều hướng tăng lên thì chọn mô hình ngay trước đó. Chú ý rằng mô hình càng phức tạp, *train error* có xu hướng càng nhỏ đi.

Hình dưới đây mô tả ví dụ phía trên với bậc của đa thức tăng từ 1 đến 8. Tập validation bao gồm 10 điểm được lấy ra từ tập training ban đầu.

Chúng ta hãy tạm chỉ xét hai đường màu lam và đỏ, tương ứng với *train error* và *validation error*. Khi bậc của đa thức tăng lên, *train error* có xu hướng giảm. Điều này dễ hiểu vì đa thức bậc càng cao, dữ liệu càng được *fit*. Quan sát đường màu đỏ, khi bậc của đa thức là 3 hoặc 4 thì *validation error* thấp, sau đó tăng dần lên. Dựa vào *validation error*, ta có thể xác định được bậc cần chọn là 3 hoặc 4. Quan sát tiếp đường màu lục, tương ứng với *test error*, thật là trùng hợp, với bậc bằng 3 hoặc 4, *test error* cũng đạt giá trị nhỏ nhất, sau đó tăng dần lên. Vậy cách làm này ở đây đã tỏ ra hiệu quả.

Việc không sử dụng *test data* khi lựa chọn mô hình ở trên nhưng vẫn có được kết quả khả quan vì ta giả sử rằng *validation data* và *test data* có chung một đặc điểm nào đó. Và khi cả hai đều là *unseen data*, *error* trên hai tập này sẽ tương đối giống nhau.

Nhắc lại rằng, khi bậc nhỏ (bằng 1 hoặc 2), cả ba error đều cao, ta nói mô hình bị *underfitting*.

15.2.2 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 tập training ra làm dữ liệu validation, phần dữ liệu còn lại của tập training là không đủ để xây dựng mô hình. Lúc này, tập validation phải thật nhỏ để giữ được lượng dữ liệu cho training đủ lớn. Tuy nhiên, một vấn đề khác nảy sinh. Khi tập validation quá nhỏ, hiện tượng overfitting lại có thể xảy ra với tập training 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 tập validation là nhỏ nhưng chất lượng mô hình được đánh giá trên nhiều tập *validation* khác nhau. Một cách thường đường sử dụng là chia tập training ra k tập con không có phần tử chung, có kích thước gần bằng nhau. Tại mỗi lần kiểm thử , được gọi là *run*, một trong số k tập con được lấy ra làm *validata set*. Mô hình 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 *train 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 tập *training* ban đầu, tức mỗi tập con có đúng 1 phần tử, ta gọi kỹ thuật này là **leave-one-out**.

Sklearn hỗ trợ rất nhiều phương thức cho phân chia dữ liệu và tính toán *scores* của các mô hình. Bạn đọc có thể xem thêm tại Cross-validation: evaluating estimator performance.

15.3 Regularization

Một nhược điểm lớn của *cross-validation* là số lượng *training runs* tỉ lệ thuận với k. Điều đáng nói là mô hình polynomial như trên chỉ có một tham số cần xác định là bậc của đa thức. Trong các bài toán Machine Learning, 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. Như vậy, việc chỉ xây dựng một mô hình thôi cũng là đã rất phức tạp rồi. Có một cách giúp số mô hình cần huấn luyện giảm đi nhiều, thậm chí chỉ một mô hình. Cách này có tên gọi chung là *regularization*.

Regularization, một cách cơ bản, là thay đổi mô hình một chút để tránh overfitting trong khi vẫn giữ được tính tổng quát của nó (tính tổng quát là tính mô tả được nhiều dữ liệu, trong cả tập training và test). Một cách cụ thể hơn, ta sẽ tìm cách *di chuyển* nghiệm của bài toán tối ưu hàm mất mát tới một điểm gần nó. Hướng di chuyển sẽ là hướng làm cho mô hình *ít phức tạp hơn* mặc dù giá trị của hàm mất mát có tăng lên một chút.

Một kỹ thuật rất đơn giản là *early stopping*.

15.3.1 Early Stopping

Trong nhiều bài toán Machine Learning, chúng ta cần sử dụng các thuật toán lặp để tìm ra nghiệm, ví dụ như Gradient Descent. Nhìn chung, hàm mất mát giảm dần khi số vòng lặp tăng lên. Early stopping tức dừng thuật toán trước khi hàm mất mát đạt giá trị quá nhỏ, giúp tránh overfitting.

Vậy dừng khi nào là phù hợp?

Một kỹ thuật thường được sử dụng là tách từ training set ra một tập validation set như trên. Sau một (hoặc một số, ví dụ 50) vòng lặp, ta tính cả *train error* và *validation error*, đến khi *validation error* có chiều hướng tăng lên thì dừng lại, và quay lại sử dụng mô hình tương ứng với điểm và *validation error* đạt giá trị nhỏ.

Hình trên đây 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.

15.3.2 Thêm số hạng vào hàm mất mát

Kỹ thuật regularization phổ biến nhất là thêm vào hàm mất mát một số hạng nữa. Số hạng này thường dùng để đánh giá độ phức tạp của mô hình. Số hạng này càng lớn, thì mô hình càng phức tạp. *Hàm mất mát mới* này thường được gọi là **regularized loss function**, thường được định nghĩa như sau:

$$J_{\text{reg}}(\theta) = J(\theta) + \lambda R(\theta)$$

Nhắc lại rằng θ được dùng để ký hiệu các biến trong mô hình, chẳng hạn như các hệ số $\mathbf{w}$ trong Neural Networks. $J(\theta)$ ký hiệu cho hàm mất mát (*loss function*) và $R(\theta)$ là số hạng *regularization*. λ thường là một số dương để cân bằng giữa hai đại lượng ở vế phải.

Việc tối thiểu *regularized loss function*, nói một cách tương đối, đồng nghĩa với việc tối thiểu cả *loss function* và số hạng *regularization*. Tôi dùng cụm "nói một cách tương đối" vì nghiệm của bài toán tối ưu *loss function* và **regularized loss function** là khác nhau. Chúng ta vẫn mong muốn rằng sự khác nhau này là nhỏ, vì vậy tham số regularization (*regularizaton parameter*) λ thường được chọn là một số nhỏ để biểu thức regularization không làm giảm quá nhiều chất lượng của nghiệm.

Với các mô hình Neural Networks, một số kỹ thuật regularization thường được sử dụng là:

15.3.3 l_2 regularization

Trong kỹ thuật này:

$$R(\mathbf{w}) = \|\mathbf{w}\|_2^2$$

tức norm 2 của hệ số. *Nếu bạn đọc chưa quen thuộc với khái niệm norm, bạn được khuyến khích đọc* phần phụ lục này.

Hàm số này có một vài đặc điểm đang lưu ý:

- Thứ nhất, $\|\mathbf{w}\|_2^2$ là một hàm số *rất mượt*, đạo hàm của nó đơn giản là $\mathbf{w}$, vì vậy đạo hàm của *regularized loss function* cũng rất dễ tính, chúng ta có thể hoàn toàn dùng các phương pháp dựa trên gradient để cập nhật nghiệm. Cụ thể:

$$\frac{\partial J_{\text{reg}}}{\partial \mathbf{w}} = \frac{\partial J}{\partial \mathbf{w}} + \lambda \mathbf{w}$$

- Thứ hai, việc tối thiểu $\|\mathbf{w}\|_2^2$ đồng nghĩa với việc khiến cho các giá trị của hệ số $\mathbf{w}$ trở nên nhỏ gần với 0. Với Polynomial Regression, việc các hệ số này nhỏ có thể giúp các hệ số ứng với các số hạng bậc cao là nhỏ, giúp tránh overfitting. Với Multi-layer Pereceptron, việc các hệ số này nhỏ giúp cho nhiều hệ số trong các ma trận trọng số là nhỏ. Điều này tương ứng với việc số lượng các hidden units *hoạt động* (khác không) là nhỏ, cũng giúp cho MLP tránh được hiện tượng overfitting.

l_2 regularization là kỹ thuật được sử dụng nhiều nhất để giúp Neural Networks tránh được overfitting. Nó còn có tên gọi khác là **weight decay**. *Decay* có nghĩa là *tiêu biến*.

Trong Xác suất thống kê, Linear Regression với l_2 regularization được gọi là **Ridge Regression**. Hàm mất mát của *Ridge Regression* có dạng:

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$$

trong đó, số hạng đầu tiên ở vế phải chính là hàm mất mát của Linear Regression. Số hạng thứ hai chính là phần regularization.

Ví dụ về Weight Decay với MLP

Chúng ta sử dụng mô hình MLP giống như bài trước nhưng dữ liệu có khác đi đôi chút.

```
# To support both python 2 and python 3
from __future__ import division, print_function, unicode_literals
import math
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(4)

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)
```

Dữ liệu được tạo là ba cụm tuân theo phân phối chuẩn có tâm ở `[[-1, -1], [1, -1], [0, 1]]`.

Trong ví dụ này, chúng ta sử dụng số hạng regularization:

$$\lambda R(\mathbf{W}) = \lambda \sum_{l=1}^{L} \|\mathbf{W}^{(l)}\|_F^2$$

với $\|.\|_F$ là Frobenius norm, là căn bậc hai của tổng bình phường các phần tử của ma trận.

(Bạn đọc được khuyến khích đọc bài MLP để hiểu các ký hiệu).

Chú ý rằng weight decay ít khi được áp dụng lên biases. Tôi thay đổi tham số regularization λ và nhận được kết quả như sau:

Khi $\lambda = 0$, tức không có regularization, ta nhận thấy gần như toàn bộ dữ liệu trong tập training được phân lớp đúng. Việc này khiến cho các class bị phân làm nhiều mảnh không được tự nhiên. Khi $\lambda = 0.001$, vẫn là một số nhỏ, các đường phân chia trông tự nhiên hơn,

nhưng lớp màu xanh lam vẫn bị chia làm hai bởi lớp màu xanh lục. Đây chính là biểu hiện của overfitting.

Khi λ tăng lên, tức sự ảnh hưởng của regularization tăng lên (xem hàng dưới), đường ranh giới giữa các lớp trở lên tự nhiên hơn. Nói cách khác, với λ đủ lớn, weight decay có tác dụng hạn chế overfitting trong MLP.

Bạn đọc hãy thử vào trong Source code, thay $\lambda = 1$ bằng cách thay dòng cuối cùng:

```
mynet(1)
```

rồi chạy lại toàn bộ code, xem các đường phân lớp trông như thế nào. Gợi ý: *underfitting*.

Khi λ quá lớn, tức ta xem phần *regularization* quan trọng hơn phần *loss fucntion*, một hiện tượng xấu xảy ra là các phần tử của $\mathbf{w}$ tiến về 0 để thỏa mãn regularization là nhỏ.

Sklearn có cung cấp rất nhiều chức năng cho MLP, trong đó ta có thể lựa chọn số lượng hidden layers và số lượng hidden units trong mỗi layer, activation functions, weight decay, learning rate, hệ số momentum, nesterovs_momentum, có early stopping hay không, lượng dữ liệu được tách ra làm validation set, và nhiều chức năng khác.

15.3.4 Tikhonov regularization

$$\lambda R(\mathbf{w}) = \|\Gamma\mathbf{w}\|_2^2$$

Với Γ (viết hoa của gamma) là một ma trận. Ma trận Γ hay được dùng nhất là ma trận đường chéo. Nhận thấy rằng l_2 regularization chính là một trường hợp đặc biệt của Tikhonov regularization với $\Gamma = \lambda\mathbf{I}$ với $\mathbf{I}$ là ma trận đơn vị (*the identity matrix*), tức các phần tử trên đường chéo của Γ là như nhau.

Khi các phần tử trên đường chéo của Γ là khác nhau, ta có một phiên bản gọi là *weighted* l_2 *regularization*, tức đánh trọng số khác nhau cho mỗi phần tử trong $\mathbf{w}$. Phần tử nào càng bị đánh trọng số cao thì nghiệm tương ứng càng nhỏ (để đảm bảo rằng hàm mất mát là nhỏ). Với Polynomial Regression, các phần tử ứng với hệ số bậc cao sẽ được đánh trọng số cao hơn, khiến cho xác suất để chúng gần 0 là lớn hơn.

15.3.5 Regularizers for sparsity

Trong nhiều trường hợp, ta muốn các hệ số *thực sự* bằng 0 chứ không phải là *nhỏ gần 0* như l_2 regularization đã làm phía trên. Lúc đó, có một regularization khác được sử dụng, đó là l_0 regularization:

$$R(\mathbf{W}) = \|\mathbf{w}\|_0$$

Norm 0 không phải là một norm thực sự mà là giả norm. (Bạn được khuyến khích đọc thêm về norms (chuẩn)). Norm 0 của một vector là số các phần tử khác không của vector đó. Khi norm 0 nhỏ, tức rất nhiều phần tử trong vector đó bằng 0, ta nói vector đó là *sparse*.

Việc giải bài toán tối thiểu norm 0 nhìn chung là khó vì hàm số này không *convex*, không liên tục. Thay vào đó, norm 1 thường được sử dụng:

$$R(\mathbf{W}) = \|\mathbf{w}\|_1 = \sum_{i=0}^{d} \|w_i\|$$

Norm 1 là tổng các trị tuyệt đối của tất cả các phần tử. Người ta đã chứng minh được rằng tối thiểu norm 1 sẽ dẫn tới nghiệm có nhiều phần tử bằng 0. Ngoài ra, vì norm 1 là một *norm thực sự* (proper norm) nên hàm số này là *convex*, và hiển nhiên là liên tục, việc giải bài toán này dễ hơn việc giải bài toán tối thiểu norm 0. Về l_1 regularization, bạn đọc có thể đọc thêm trong lecture note này. Việc giải bài toán l_1 regularization nằm ngoài mục đích của tôi trong bài viết này. Tôi hứa sẽ quay lại phần này sau. (Vì đây là phần chính trong nghiên cứu của tôi).

Trong Thống Kê, việc sử dụng l_1 regularization còn được gọi là LASSO) (Least Absolute Shrinkage and Selection Operator)).

Khi cả l_2 và l_1 regularization được sử dụng, ta có mô hình gọi là Elastic Net Regression.

15.3.6 Regularization trong sklearn

Trong sklearn, ví dụ Logistic Regression, bạn cũng có thể sử dụng các l_1 và l_2 regularizations bằng cách khai báo biến `penalty='l1'` hoặc `penalty = 'l2'` và biến `C`, trong đó `C` là *nghịch đảo* của λ. Trong các bài trước khi chưa nói về Overfitting và Regularization, tôi có sử dụng `C = 1e5` để chỉ ra rằng λ là một số rất nhỏ.

15.4 Các phương pháp khác

Ngoài các phương pháp đã nêu ở trên, với mỗi mô hình, nhiều phương pháp tránh overfitting khác cũng được sử dụng. Điển hình là Dropout trong Deep Neural Networks mới được đề xuất gần đây. Một cách ngắn gọn, dropout là một phương pháp *tắt* ngẫu nhiên các units trong Networks. *Tắt* tức cho các unit giá trị bằng không và tính toán feedforward và backpropagation bình thường trong khi training. Việc này không những giúp lượng tính toán giảm đi mà còn làm giảm việc overffitng. Tôi xin được quay lại vấn đề này nếu có dịp nói sâu về Deep Learning trong tương lai.

Bạn đọc có thể tìm đọc thêm với các từ khóa: pruning) (tránh overftting trong Decision Trees), VC dimension (đo độ phức tạp của mô hình, độ phức tạp càng lớn thì càng dễ bị overfitting).

15.5 Tóm tắt nội dung

- Một mô hình mô tốt là mộ mô hình có *tính tổng quát*, tức mô tả được dữ liệu cả trong lẫn ngoài tập training. Mô hình chỉ mô tả tốt dữ liệu trong tập training được gọi là **overfitting**.
- Để tránh overfitting, có rất nhiều kỹ thuật được sử dụng, điển hình là **cross-validation** và **regularization**. Trong Neural Networks, **weight decay** và **dropout** thường được dùng.

15.6 Tài liệu tham khảo

[1] Overfitting - Wikipedia

[2] Cross-validation - Wikipedia)

[3] Pattern Recognition and Machine Learning

[4] Krogh, Anders, and John A. Hertz. "A simple weight decay can improve generalization." NIPS. Vol. 4. 1991.

[5] Srivastava, Nitish, et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" Journal of Machine Learning Research 15.1 (2014): 1929-1958.

Index