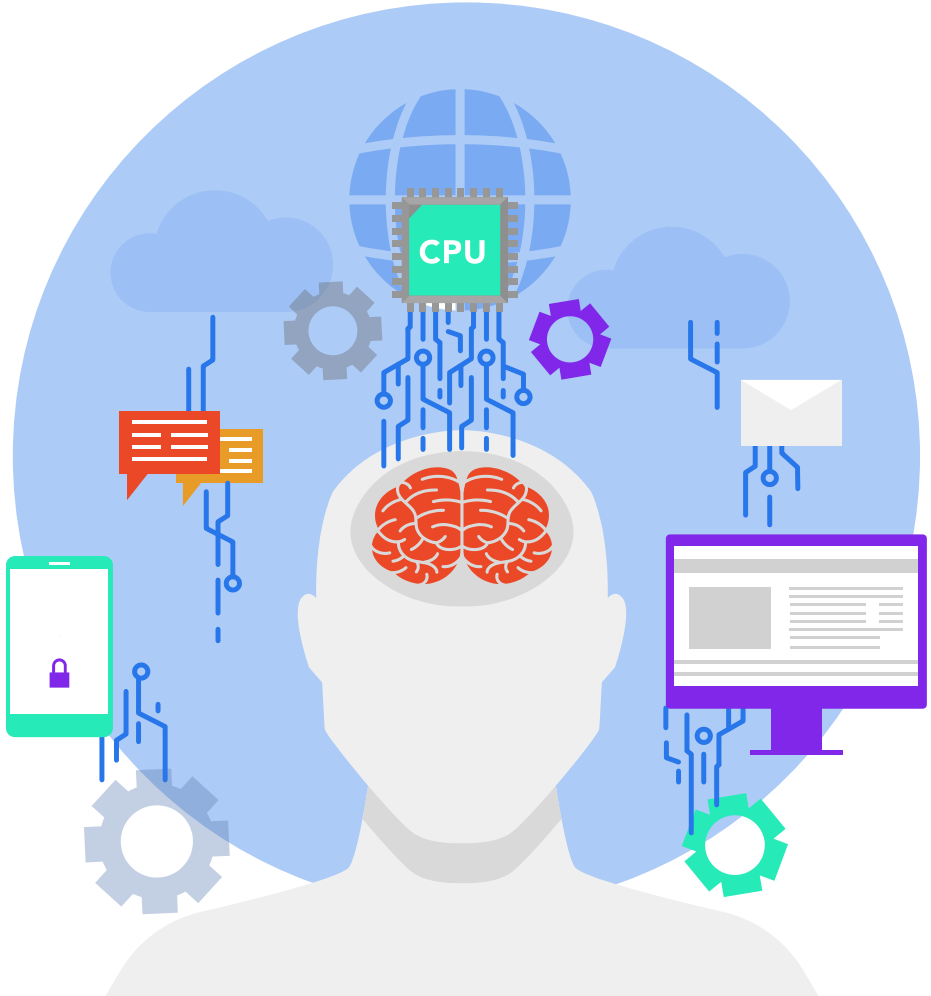
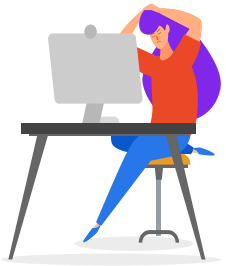


Chương 4: Các máy Vector hỗ trợ (SVM) và kỹ thuật Boosting

4.1 Ý tưởng chính



4.1.1 Khoảng cách từ một điểm tới một siêu mặt phẳng

Trong không gian 2 chiều, ta biết rằng khoảng cách từ một điểm có toạ độ **(x0,y0)** tới *đường thẳng* có phương trình $w_1 x_0 + w_2 y_0 + b = 0$ được xác định bởi:

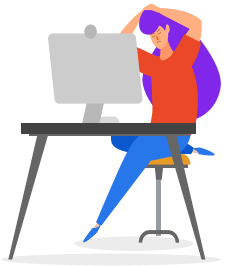
$$\frac{|w_1 x_0 + w_2 y_0 + b|}{\sqrt{w_1^2 + w_2^2}}$$

Trong không gian ba chiều, khoảng cách từ một điểm có toạ độ **(x0,y0,z0)** tới một *mặt phẳng* có phương trình $w_1 x + w_2 y + w_3 z + b = 0$ được xác định bởi:

$$\frac{|w_1 x_0 + w_2 y_0 + w_3 z_0 + b|}{\sqrt{w_1^2 + w_2^2 + w_3^2}}$$

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

4.1 Ý tưởng chính



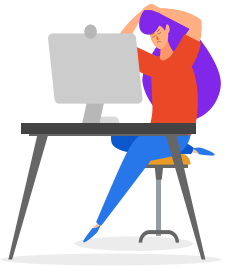
4.1.1 Khoảng cách từ một điểm tới một siêu mặt phẳng

Việc này có thể được tổng quát lên không gian nhiều chiều: Khoảng cách từ một điểm (vector) có toạ độ x_0 tới *siêu mặt phẳng* (*hyperplane*) có phương trình $w^T x + b = 0$ được xác định bởi:

$$\frac{|w^T x_0 + b|}{||w||_2}$$

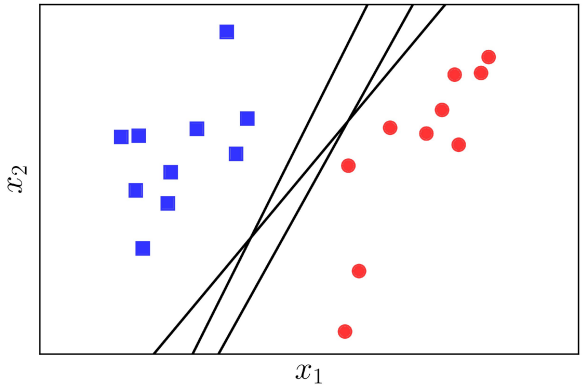
Với $||w||_2 = \sqrt{\sum_{i=1}^{d} w_i^2}$ với d là số chiều của không gian

4.1 Ý tưởng chính

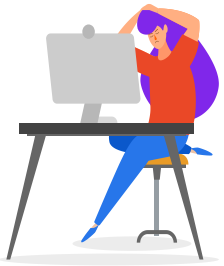


4.1.2 Nhắc lại – bài toán phân chia hai classes

Giả sử rằng có hai class khác nhau được mô tả bởi các điểm trong không gian nhiều chiều, hai classes này *linearly separable*, tức tồn tại một siêu phẳng phân chia chính xác hai classes đó. Hãy tìm một siêu mặt phẳng phân chia hai classes đó, tức tất cả các điểm thuộc một class nằm về cùng một phía của siêu mặt phẳng đó và ngược phía với toàn bộ các điểm thuộc class còn lại. Chúng ta đã biết rằng, thuật toán PLA có thể làm được việc này nhưng nó có thể cho chúng ta vô số nghiệm như Hình 1 dưới đây:

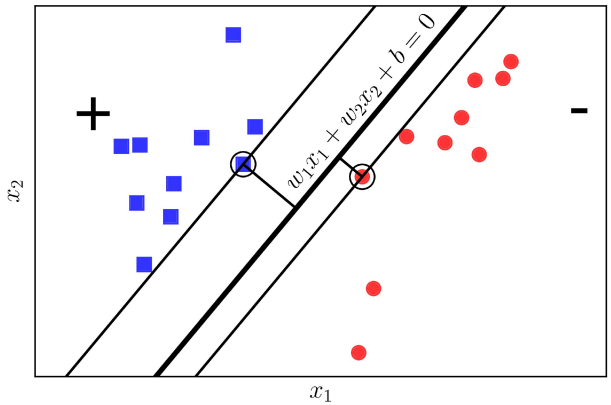


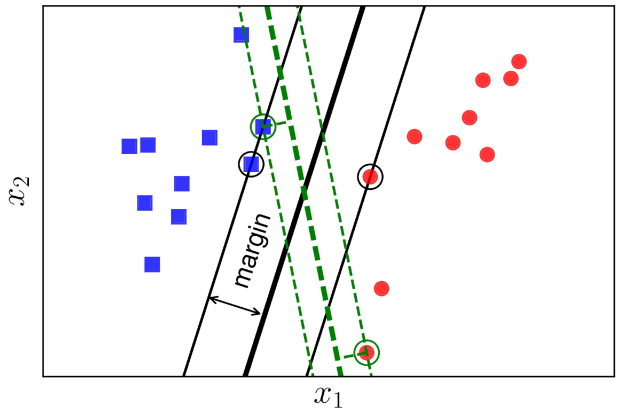
4.1 Ý tưởng chính



4.1.2 Nhắc lại – bài toán phân chia hai classes

Câu hỏi đặt ra: trong vô số các mặt phân chia đó, đâu là mặt phân chia tốt nhất *theo một tiêu chuẩn nào đó*? Trong ba đường thẳng minh họa trong Hình 1 phía trên, có hai đường thẳng khá *lệch* về phía class hình tròn đỏ. Điều này có thể khiến cho lớp màu đỏ *không vui vì lãnh thổ xem ra bị lấn nhiều quá*. Liệu có cách nào để tìm được đường phân chia mà cả hai classes đều cảm thấy *công bằng* và *hạnh phúc* nhất hay không?
Chúng ta cần tìm một tiêu chuẩn để đo sự *hạnh phúc* của mỗi class. Xem hình 2:





4.1 Ý tưởng chính

4.1.2 Nhắc lại – bài toán phân chia hai classes

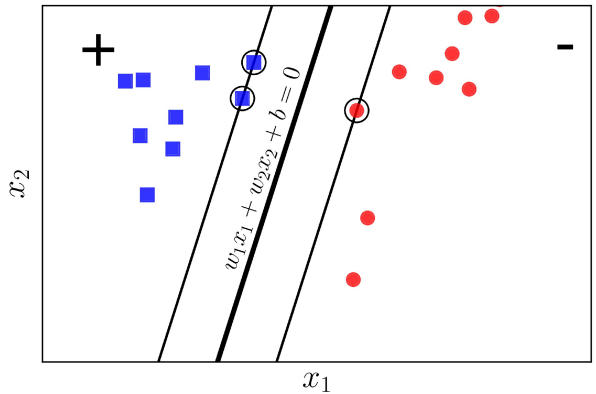
- Nếu ta định nghĩa "mức độ hạnh phúc" của một lớp là khoảng cách từ điểm gần nhất của lớp đó tới đường phân chia, thì trong Hình 2 (trái), lớp tròn đỏ **kém hạnh phúc** hơn vì đường phân chia gần nó hơn so với lớp vuông xanh. Để **công bằng**, ta cần một đường phân chia sao cho khoảng cách từ các điểm gần nhất của mỗi lớp tới đường là **bằng nhau** – đó chính là **margin (lề)**.
- Tuy nhiên, **công bằng chưa đủ**, ta còn muốn **văn minh** – nghĩa là cả hai lớp đều **có khoảng cách lớn** đến đường phân chia. Trong Hình 2 (phải), giữa hai đường phân chia (nét liền đen và nét đứt lục), rõ ràng đường **nét liền màu đen** tạo ra **margin rộng hơn**, giúp cả hai lớp "hạnh phúc" hơn.
- Một **margin rộng** giúp phân lớp **rõ ràng và ổn định** hơn – đây chính là lợi thế của **SVM** so với **Perceptron**, vốn chỉ tạo ra một ranh giới mà không tối ưu khoảng cách.

➡ Bài toán tối ưu trong *Support Vector Machine* (SVM) chính là bài toán đi tìm đường phân chia sao cho *margin* là lớn nhất.

4.2 Xây dựng bài toán tối ưu cho SVM

- Giả sử rằng các cặp dữ liệu của *training set* là (x_1, y_1), (x_2, y_2),..., (x_n, y_n) với vector $x_i \in R^d$ thể hiện *đầu vào* của một điểm dữ liệu và yi là *nhãn* của điểm dữ liệu đó. d là số chiều của dữ liệu và N là số điểm dữ liệu. Giả sử rằng *nhãn* của mỗi điểm dữ liệu được xác định bởi $y_i = 1$ (class 1) hoặc $x_i = -1$ (class 2) giống như trong PLA.

- Xét trường hợp trong không gian hai chiều dưới đây. *Không gian hai chiều để các bạn dễ hình dung, các phép toán hoàn toàn có thể được tổng quát lên không gian nhiều chiều.*



4.2 Xây dựng bài toán tối ưu cho SVM

- Giả sử rằng các điểm vuông xanh thuộc class 1, các điểm tròn đỏ thuộc class -1 và mặt $w^T x + b = w_1 x_1 + w_2 x_2 + b = 0$ là mặt phân chia giữa hai classes (Hình 3). Hơn nữa, class 1 nằm về *phía dương*, class -1 nằm về *phía âm* của mặt phân chia. Nếu ngược lại, ta chỉ cần đổi dấu của **w** và b. Chú ý rằng chúng ta cần đi tìm các hệ số **w** và b.
- Ta quan sát thấy một điểm quan trọng sau đây: với cặp dữ liệu (x_n, y_n) bất kỳ, khoảng cách từ điểm đó tới mặt phân chia là:

$$\frac{y_n(w^T x_n + b)}{||w||_2}$$

- Điều này có thể dễ nhận thấy vì theo giả sử ở trên, y_n luôn cùng dấu với *phía* của x_n. Từ đó suy ra y_n cùng dấu với $(w^T x_n + b)$, và tử số luôn là 1 số không âm.
- Với mặt phần chia như trên, *margin* được tính là khoảng cách gần nhất từ 1 điểm tới mặt đó (bất kể điểm nào trong hai classes):

$$margin = \min_n \frac{y_n(w^T x_n + b)}{||w||_2}$$

4.2 Xây dựng bài toán tối ưu cho SVM

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

$$(w, b) = \arg\max_{w,} \left\{ \min_{n} \frac{y_n (w^T x_n + b)}{||w||_2} \right\} = \arg\max_{w,} \left\{ \frac{1}{||w||_2} \min_{n} y_n (w^T x_n + b) \right\} \quad (1)$$

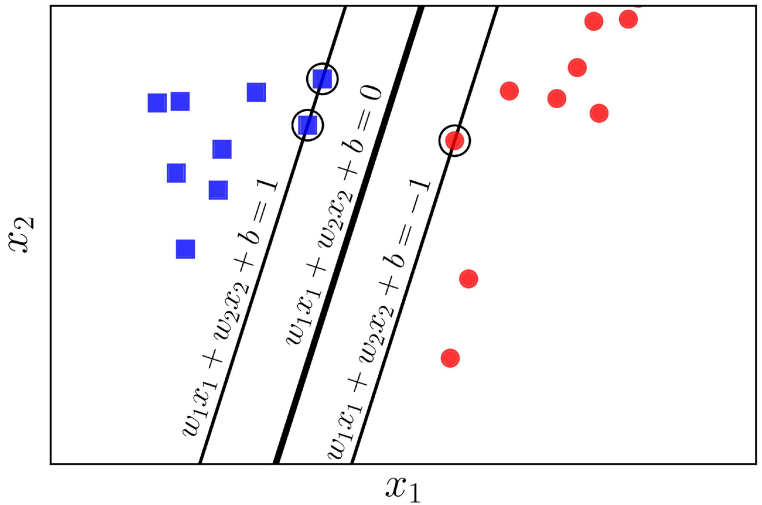
- Việc giải trực tiếp bài toán này sẽ rất phức tạp, nhưng các bạn sẽ thấy có cách để đưa nó về bài toán đơn giản hơn.

4.2 Xây dựng bài toán tối ưu cho SVM

Quan trọng nhất là nếu ta thay vector hệ số ***w*** bởi ***kw*** và ***b*** bởi ***kb*** trong đó ***k*** là một hằng số dương thì mặt phân chia không thay đổi, tức khoảng cách từ từng điểm đến mặt phân chia không đổi, tức *margin* không đổi. Dựa trên tính chất này, ta có thể giả sử:

$$y_n(w^T x_n + b) = 1$$

với những điểm nằm gần mặt phân chia nhất như Hình 4 dưới đây:



4.2 Xây dựng bài toán tối ưu cho SVM

Như vậy, với mọi n, ta có:

$$y_n(w^T x_n + b) \geq 1$$

Vậy bài toán tối ưu (1) có thể đưa về bài toán tối ưu có ràng buộc sau đây:

$$(w, b) = \arg max \frac{1}{2} ||w||_2^2$$

$$\text{Subject to: } 1 - y_n(w^T x_n + b) \leq 0, \forall m = 1, 2, \dots, N \quad (3)$$

Ở đây, chúng ta đã lấy nghịch đảo hàm mục tiêu, bình phương nó để được một hàm khả vi, và nhân với 12 để biểu thức đạo hàm đẹp hơn.

4.2 Xây dựng bài toán tối ưu cho SVM

- Bài toán (3) là bài toán lồi:
  - Hàm mục tiêu là norm → hàm lồi.
  - Ràng buộc là hàm tuyến tính theo w,bw, bw,b → cũng là lồi.
- Hàm mục tiêu là strictly convex vì $||w||^2 = w^T I w$, I là ma trận xác định dương → nghiệm là duy nhất.
- Đây là bài toán **Quadratic Programming** (QP) → có thể giải bằng công cụ như **CVXOPT**.
- Tuy nhiên, khi số chiều ddd và số điểm NNN lớn → bài toán gốc trở nên khó giải hơn.
- Vì vậy, người ta thường giải bài toán đối ngẫu:
  - Có tính chất giúp giải hiệu quả hơn.
  - Cho phép áp dụng SVM với dữ liệu không tuyến tính (non-linearly separable) → nhờ vào kernel trick.

4.2 Xây dựng bài toán tối ưu cho SVM

Xác định class cho một điểm dữ liệu mới:

Sau khi tìm được mặt phân cách wTx+b=0, class của bất kỳ một điểm nào sẽ được xác định đơn giản bằng cách:

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

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

4.3 Bài toán đối ngẫu cho SVM

Thiệt lập cơ sở cho việc xây dựng và giải bài toán đối ngẫu của SVM: Bài toán tối ưu (3) là một bài toán lồi. Chúng ta biết rằng: nếu một bài toán lồi thoả mãn tiêu chuẩn Slater thì *strong duality* thoả mãn. Và nếu *strong duality* thoả mãn thì nghiệm của bài toán chính là nghiệm của hệ điều kiên KKT.

4.3 Bài toán đối ngẫu cho SVM

4.3.1 Kiểm tra tiêu chuẩn Slater

Bước tiếp theo, chúng ta sẽ chứng minh bài toán tối ưu (3) thoả mãn điều kiện Slater. Điều kiện Slater nói rằng, nếu tồn tại w,b thoả mãn:

$$1 - y_n(w^T x_n + b) < 0, \forall n = 1,2,\dots,N$$

thì *strong duality* thoả mãn.

Việc kiểm tra này tương đối đơn giản. Vì ta biết rằng luôn luôn có một (siêu) mặt phẳng phân chia hai classes nếu hai class đó là *linearly separable*, tức bài toán có nghiệm, nên *feasible set* của bài toán tối ưu (3) phải khác rỗng. Tức luôn luôn tồn tại cặp (w_0, b_0) sao cho:

$$1 - y_n(w_0^T x_n + b_0) \leq 0, \forall n = 1,2,\dots,N$$

$$\leftrightarrow 2 - y_n(2w_0^T x_n + 2b_0) \leq 0, \forall n = 1,2,\dots,N$$

Vậy chỉ cần chọn $w_1 = 2w_0$ và $b_1 = 2b_0$, ta sẽ có:

$$1 - y_n(w_1^T x_n + b_1) \leq -1 < 0, \forall n = 1,2,\dots,N$$

4.3 Bài toán đối ngẫu cho SVM

4.3.2 Lagrangian của bài toán SVM

Largrangian của bài toán (3) là:

$$\mathcal{L}(w, b, \lambda) = \frac{1}{2}||w||_2^2 + \sum_{n=1}^{N} \lambda_n \left(1 - y_n(w^T x_n + b)\right) \quad (4)$$

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

4.3 Bài toán đối ngẫu cho SVM

4.3.3 Hàm đối ngẫu Lagrange

Hàm đối ngẫu Lagrange được định nghĩa là:

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

Với $\lambda \geq 0$

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

$$\frac{\partial \mathcal{L}(w, b, \lambda)}{\partial w} = w - \sum_{n=1}^{N} \lambda_n y_n x_n = 0 \Rightarrow w = \sum_{n=1}^{N} \lambda_n y_n x_n \qquad (5)$$

$$\frac{\partial \mathcal{L}(w, b, \lambda)}{\partial w} = -\sum_{n=1}^{N} \lambda_n y_n = 0 \qquad (6)$$

4.3 Bài toán đối ngẫu cho SVM

4.3.3 Hàm đối ngẫu Lagrange

Thay (5) và (6) vào (4) ta thu được g(λ)(*phần này tôi rút gọn, coi như một bài tập nhỏ cho bạn nào muốn hiểu sâu*):

$$g(\lambda) = \sum_{n=1}^{N} \lambda_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \lambda_n \lambda_m y_n y_m x_n^T x_m \qquad (7)$$

Đây là hàm số quan trọng nhất trong SVM

4.3 Bài toán đối ngẫu cho SVM

4.3.3 Hàm đối ngẫu Lagrange

Ví dụ:

Xét ma trận:

$$V = [y_1 x_1,\ y_2 x_2, \dots, y_n x_n]$$

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

$$g(\lambda) = -\frac{1}{2}\lambda^T V^T V\lambda + 1^T\lambda \quad (8)$$

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

$$\lambda^T K\lambda = \lambda^T V^T V\lambda = ||V\lambda||_2^2 \geq 0$$

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

4.3 Bài toán đối ngẫu cho SVM

4.3.3 Bài toán đối ngẫu Lagrange

- Từ trên, kết hợp hàm đối ngẫu Lagrange và các điều kiện ràng buộc của λ, ta sẽ thu được bài toán đối ngẫu Lagrange:

$$\lambda = \arg\max_{\lambda} g(\lambda)$$

$$\text{Subject to: } \lambda \geq 0 \quad (9)$$

$$\sum_{n=1}^{N} \lambda_n y_n = 0$$

- Ràng buộc thứ hai được lấy từ (6).

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

- Bài toán này cũng được là một Quadratic Programming và cũng có thể được giải bằng các

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Bài toán đối ngẫu Lagrange

- Trong bài toán đối ngẫu, ta cần tìm **N** tham số (bằng số điểm dữ liệu), trong khi bài toán gốc chỉ có **d+1** tham số (số chiều dữ liệu cộng 1). Khi số dữ liệu lớn hơn số chiều, giải bài toán đối ngẫu có thể tốn thời gian hơn nếu dùng phương pháp **QP** thông thường.
- Tuy nhiên, điều hấp dẫn của bài toán đối ngẫu là ở Kernel SVM, giúp xử lý các bài toán dữ liệu không tuyến tính. Ngoài ra, nhờ điều kiện KKT, SVM có thể được giải bằng những cách hiệu quả hơn.

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Điều kiện KKT

Quay trở lại bài toán, vì đây là một bài toán lồi và *strong duality* thoả mãn, nghiệm của bài toán sẽ thoả mãn hệ điều kiện KKT sau đây với biến số là w,b và λ:

$$1 - y_n(w^T x_n + b) \leq 0, \forall n = 1,2,\dots,N \quad (10)$$

$$\lambda_n \geq 0, \forall n = 1,2,\dots,N$$

$$\lambda_n\big(1 - y_n(w^T x_n + b)\big) = 0, \forall n = 1,2,\dots,N \quad (11)$$

$$w = \sum_{n=1}^{N} \lambda_n y_n x_n \quad (12)$$

$$\sum_{n=1}^{N} \lambda_n y_n = 0 \quad (13)$$

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Điều kiện KKT

Từ điều kiện (11) ta có thể suy ra ngay, với n bất kỳ, hoặc $\lambda_n = 0$ hoặc $1 - y_n(w^T x_n + b) = 0$. Trường hợp thứ hai chính là:

$$w^T x_n + b = y_n \quad (14)$$

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

Những điểm thoả mãn (14) chính là những điểm nằm gần mặt phân chia nhất, là những điểm được khoanh tròn trong Hình 4 phía trên. Hai đường thẳng $(w^T x_n + b) = \pm 1$ *tựa* lên các điểm thoả mãn (14). Vậy nên những điểm (vectors) thoả mãn (14) còn được gọi là các *Support Vectors*. Nguồn gốc cái tên *Support Vector Machine*.

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Điều kiện KKT

- Chỉ một số rất nhỏ điểm dữ liệu thỏa mãn điều kiện (14) → chính là support vectors.
- Mặt phân cách được xác định chỉ từ các support vectors.
- Hầu hết các hệ số λ_n bằng 0 → vector λ_n là một sparse vector.
- Vì thế, SVM là mô hình sparse (Sparse Model).
- Sparse Models thường có cách giải hiệu quả hơn so với mô hình dense (nhiều nghiệm khác 0).
- Đây là lý do thứ hai khiến bài toán đối ngẫu SVM được ưu tiên hơn so với bài toán gốc.

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Điều kiện KKT

- Với những bài toán có số điểm dữ liệu N nhỏ, ta có thể giải hệ điều kiện KKT phía trên bằng cách xét các trường hợp $\lambda_n=0$ hoặc $\lambda_n\neq0$. Tổng số trường hợp phải xét là 2^N. Với N>50 (thường là như thế) → một con số rất lớn → không khả thi.
- Sau khi tìm được λ từ bài toán (9), ta có thể suy ra được w dựa vào (12) và b dựa vào (11) và (13). Rõ ràng ta chỉ cần quan tâm tới $\lambda_n \neq 0$.
- Gọi tập hợp $S = \{n: \lambda_n \neq 0\}$ và N_S là số phần tử của tập S. Với mỗi $n \in S$, ta có:

$$1 = y_n(w^T x_n + b) \leftrightarrow b + w^T x_n = y_n$$

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

$$b = \frac{1}{N_S}\sum(y_n - w^T x_n) = \frac{1}{N_S}\sum\left(y_n - \sum \lambda_m y_m x_m^T x_n\right) \qquad (15)$$

4.3 Bài toán đối ngẫu cho SVM

4.3.4 Điều kiện KKT

Trước đó, w đã được tính bằng:

$$w = \sum_{m \in S} \lambda_m y_m x_m \quad (16)$$

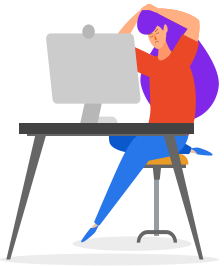
theo (12).

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

$$w^T x + b = \sum_{m \in S} \lambda_m y_m x_m^T x + \frac{1}{N_S} \sum_{n \in S} (y_n - \sum_{m \in S} \lambda_m y_m x_m^T x_n)$$

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

4.4 Lập trình tìm nghiệm cho SVM



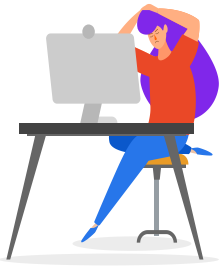
4.4.1 Tìm nghiệm theo công thức

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

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

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

4.4 Lập trình tìm nghiệm cho SVM



4.4.1 Tìm nghiệm theo công thức

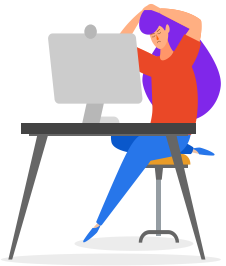
Tiếp theo, chúng ta giải bài toán (9) bằng CVXOPT:

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

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

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

4.4 Lập trình tìm nghiệm cho SVM



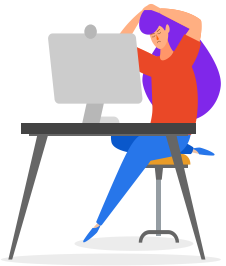
4.4.1 Tìm nghiệm theo công thức

Kết quả:

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

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

4.4 Lập trình tìm nghiệm cho SVM



4.4.1 Tìm nghiệm theo công thức

Ta đi tìm *support set* S rồi tìm nghiệm của bài toán:

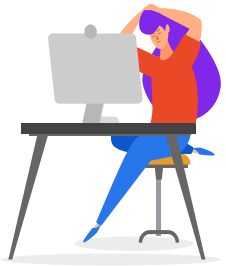
```
epsilon = 1e-6 # just a small number, greater than 1e-9
S = np.where(l > epsilon)[0]

VS = V[:, S]
XS = X[:, S]
yS = y[:, S]
lS = l[S]
# calculate w and b
w = VS.dot(lS)
b = np.mean(yS.T - w.T.dot(XS))

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

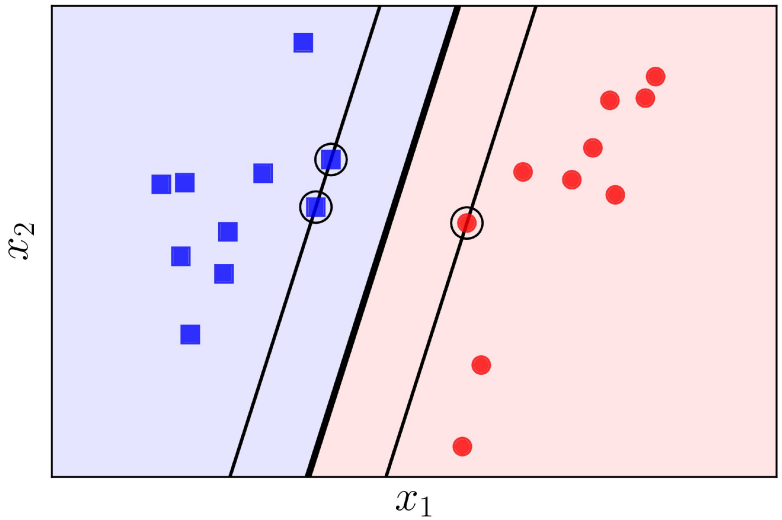
```
w =  [[-2.00984381  0.64068336]]
b =  4.66856063387
```

4.4 Lập trình tìm nghiệm cho SVM



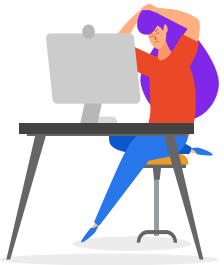
4.4.1 Tìm nghiệm theo công thức

Minh hoạ kết quả:



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

4.4 Lập trình tìm nghiệm cho SVM



4.4.2 Tìm nghiệm theo thư viện

Sử dụng hàm sklearn.svm.SVC ở đây. Các bài toán thực tế thường sử dụng thư viện libsvm được viết trên ngôn ngữ C, có API cho Python và Matlab. Kết quả này khá giống với kết quả ở phần trên

```
from sklearn.svm import SVC

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

clf.fit(X1, y1)

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

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

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.1 Phân loại văn bản

SVM là một trong những thuật toán hàng đầu cho các bài toán phân loại văn bản nhờ khả năng xử lý dữ liệu thưa (sparse data) và không gian đặc trưng chiều cao

- **Phát hiện email rác (spam):** SVM được sử dụng để phân loại email thành "rác" hoặc "không rác" dựa trên các đặc trưng như tần suất từ, cấu trúc câu hoặc metadata. Ví dụ, các đặc trưng được trích xuất từ nội dung email (sử dụng TF-IDF hoặc Bag-of-Words) được đưa vào SVM với kernel tuyến tính hoặc RBF để phân loại.
- **Phân tích cảm xúc (sentiment analysis):** SVM phân loại các bình luận, bài đánh giá thành tích cực, tiêu cực hoặc trung tính. Chẳng hạn, trong các hệ thống thương mại điện tử, SVM giúp phân tích cảm xúc khách hàng dựa trên đánh giá sản phẩm.
- **Phân loại tài liệu:** SVM được sử dụng để phân loại tài liệu vào các chủ đề (topic classification), ví dụ như phân loại tin tức thành các danh mục như thể thao, chính trị, kinh tế.

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.2 Nhận dạng hình ảnh

Trong thị giác máy tính, SVM được áp dụng để xử lý các bài toán nhận dạng và phân loại hình ảnh, đặc biệt khi kết hợp với các kỹ thuật trích xuất đặc trưng như HOG (Histogram of Oriented Gradients) hoặc SIFT (Scale-Invariant Feature Transform)

- **Nhận dạng đối tượng:** SVM được sử dụng để nhận diện các đối tượng trong ảnh, ví dụ như phát hiện xe cộ trong hệ thống giám sát giao thông. Các đặc trưng hình ảnh được trích xuất và ánh xạ vào không gian chiều cao để phân loại.
- **Phát hiện khuôn mặt:** Trong các hệ thống an ninh hoặc ứng dụng nhận diện, SVM phân loại các vùng ảnh có chứa khuôn mặt hay không. Ví dụ, các đặc trưng được trích xuất từ ảnh khuôn mặt (như Haar-like features) được đưa vào SVM để dự đoán.
- **Phân loại hình ảnh y tế:** SVM được sử dụng để phân loại hình ảnh MRI hoặc CT để phát hiện các bất thường như khối u hoặc tổn thương.

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.3 Sinh học và y học

SVM là công cụ quan trọng trong tin sinh học và y học nhờ khả năng xử lý dữ liệu phức tạp và nhiễu, thường gặp trong các tập dữ liệu sinh học.

- **Phân loại Gen:** SVM được sử dụng để phân loại các mẫu gen liên quan đến bệnh lý, ví dụ như xác định các gen liên quan đến ung thư dựa trên dữ liệu microarray.
- **Dự đoán bệnh:** Trong chẩn đoán y tế, SVM được áp dụng để dự đoán các bệnh như ung thư vú hoặc tiểu đường dựa trên dữ liệu lâm sàng hoặc hình ảnh y tế. Ví dụ, SVM có thể phân loại bệnh nhân dựa trên các đặc trưng như chỉ số khối cơ thể (BMI), mức đường huyết, hoặc đặc điểm hình ảnh từ chụp X-quang.
- **Phân tích protein:** SVM được sử dụng để dự đoán cấu trúc protein hoặc tương tác protein-protein dựa trên các đặc trưng hóa học và sinh học.

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.4 Dự đoán tài chính

Trong lĩnh vực tài chính, SVM được sử dụng để xử lý các bài toán liên quan đến dữ liệu không tuyến tính và biến động cao.

- **Dự đoán giá cổ phiếu:** SVM hồi quy (SVR) được sử dụng để dự đoán xu hướng giá cổ phiếu dựa trên dữ liệu lịch sử, các chỉ số kinh tế hoặc tâm lý thị trường.
- **Đánh giá rủi ro tín dụng:** SVM phân loại khách hàng thành các nhóm "rủi ro thấp" hoặc "rủi ro cao" dựa trên hồ sơ tín dụng, thu nhập, và lịch sử giao dịch.
- **Phát hiện gian lận tài chính:** SVM được áp dụng để phát hiện các giao dịch bất thường, ví dụ như gian lận thẻ tín dụng, bằng cách phân tích các mẫu giao dịch và phát hiện các điểm ngoại lai.

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.5 Nhận dạng chữ viết tay và các ứng dụng khác

SVM là lựa chọn phổ biến trong các bài toán nhận dạng chữ viết tay nhờ khả năng tối ưu hóa lề và xử lý dữ liệu phức tạp.

- **Nhận dạng chữ số:** Trong cơ sở dữ liệu MNIST, SVM được sử dụng để phân loại các chữ số viết tay (0-9) dựa trên các đặc trưng pixel hoặc đặc trưng được trích xuất.
- **Nhận dạng ký tự:** SVM được áp dụng trong các hệ thống nhận dạng ký tự quang học (OCR) để nhận diện chữ viết tay hoặc văn bản in trong các tài liệu.

Ứng dụng khác:

- **An ninh mạng:** SVM được sử dụng để phát hiện xâm nhập (intrusion detection) trong mạng máy tính bằng cách phân loại lưu lượng mạng thành "bình thường" hoặc "bất thường".
- **Dự báo thời tiết:** SVM có thể được sử dụng để dự đoán các hiện tượng thời tiết như mưa hoặc bão dựa trên dữ liệu khí tượng.
- **Khuyến nghị sản phẩm:** Trong các hệ thống khuyến nghị, SVM có thể được sử dụng để phân loại sở thích người dùng dựa trên hành vi mua sắm hoặc tương tác.

4.5 Ứng dụng của Máy Vector hỗ trợ SVM

4.5.3 Sinh học và y học

SVM là công cụ quan trọng trong tin sinh học và y học nhờ khả năng xử lý dữ liệu phức tạp và nhiễu, thường gặp trong các tập dữ liệu sinh học.

- **Phân loại Gen:** SVM được sử dụng để phân loại các mẫu gen liên quan đến bệnh lý, ví dụ như xác định các gen liên quan đến ung thư dựa trên dữ liệu microarray.
- **Dự đoán bệnh:** Trong chẩn đoán y tế, SVM được áp dụng để dự đoán các bệnh như ung thư vú hoặc tiểu đường dựa trên dữ liệu lâm sàng hoặc hình ảnh y tế. Ví dụ, SVM có thể phân loại bệnh nhân dựa trên các đặc trưng như chỉ số khối cơ thể (BMI), mức đường huyết, hoặc đặc điểm hình ảnh từ chụp X-quang.
- **Phân tích protein:** SVM được sử dụng để dự đoán cấu trúc protein hoặc tương tác protein-protein dựa trên các đặc trưng hóa học và sinh học.

4.6 Kỹ thuật Boosting

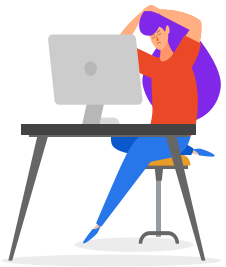
4.6.1 Nguyên lý hoạt động

Kỹ thuật Boosting là một phương pháp học máy thuộc nhóm học tập hợp (ensemble learning), nhằm cải thiện hiệu suất của các mô hình học máy bằng cách kết hợp nhiều mô hình yếu (weak learners) để tạo ra một mô hình mạnh (strong learner).

Nguyên lý hoạt động: Boosting tập trung vào việc cải thiện các dự đoán sai của các mô hình yếu bằng cách gán trọng số cao hơn cho các mẫu dữ liệu bị phân loại sai trong các vòng lặp huấn luyện. Các mô hình yếu (thường là cây quyết định) được huấn luyện tuần tự, và mỗi mô hình cố gắng sửa lỗi của các mô hình trước đó.

Các thuật toán Boosting phổ biến:

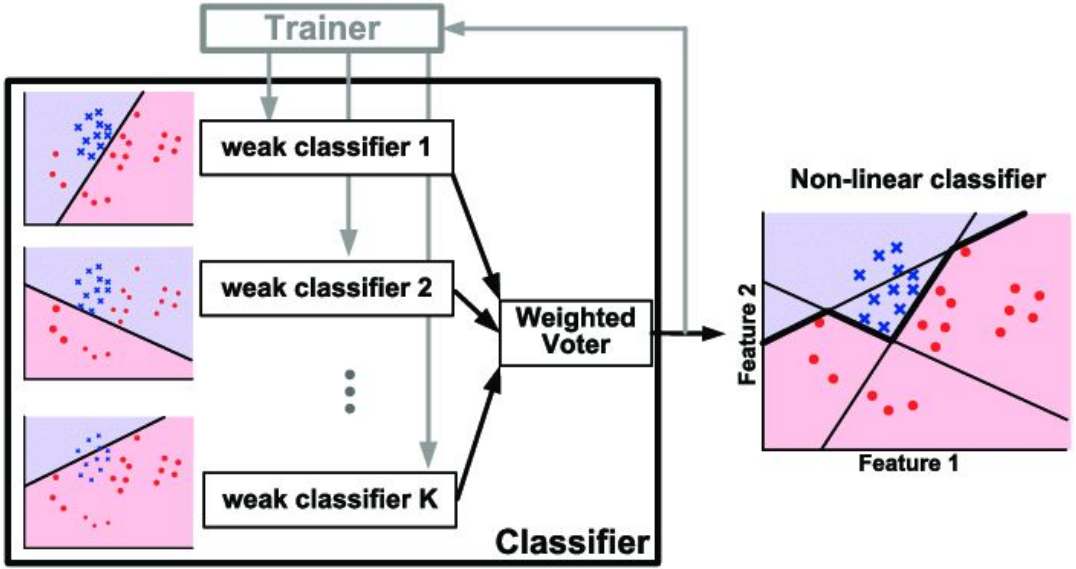
- **AdaBoost (Adaptive Boosting)**
- **Gradient Boosting**
- **XGBoost (Extreme Gradient Boosting)**
- **LightGBM**

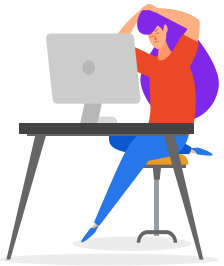


4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

AdaBoost, tên đầy đủ là Adaptive Boosting, là thuật toán thuộc nhánh Boosting trong Ensemble learning. Với ý tưởng đơn giản là sử dụng các cây quyết định (1 gốc, 2 lá) để đánh trọng số cho các điểm dữ liệu, từ đó tối thiểu hóa trọng số các điểm bị phân loại sai (trọng số lớn), để tăng hiệu suất của mô hình





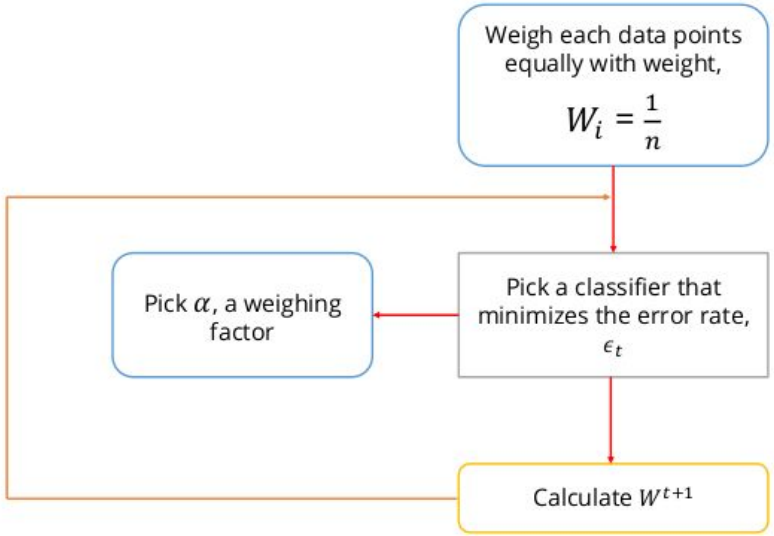
4.6 Kỹ thuật Boosting

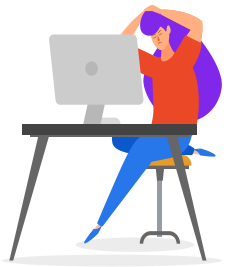
4.6.2 AdaBoost

Giải thuật AdaBoost:

Có nghĩa là với:

- Tập dữ liệu $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ với $y_i \in \{-1,1\}, i \in \{1,2, \dots, n\}$
- Trọng số các điểm dữ liệu tại weak leaner thứ t:$w_1^t, w_2^t, \dots, w_n^t, i \in \{1,2, \dots, n\}$
- Weak-learners h: $x \rightarrow \{-1, 1\}$
- Error function: $E(f(x), y, i) = e^{-y_i f(x_i)}$ trong đó $f(x_i) = \propto h(x_i)$
- Output: $H_t(x)$ còn được gọi là strong learner tại thời điểm t





4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Triển khai thuật toán:

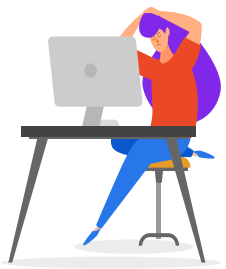
B1: Khởi tạo weights cho từng input: $w_i^1 = \frac{1}{n}, i \in \{1, 2, ..., n\} \rightarrow \sum_i w_i^1 = 1$

B2: Với mỗi vòng lặp $t \in \{1, 2, ..., T\}$:

- Tìm weak-learners $h_t(x)$ để tối thiểu hóa tổng error của các điểm bị phân loại sai, $E = \sum_{y_i \# h_t(x_i)} w_i^t$
- Tỉ lệ lỗi của weak leaners: $\varepsilon_t = \frac{\sum_{y_i \# h_t(x_i)} w_i^t}{\sum_{i=1}^n w_i^t} = \sum_{y_i \# h_t(x_i)} w_i^t$, vì $\sum_{i=1}^n w_i^t = 1$
- Gán trọng số cho weak-learners giá trị $\alpha_t = \frac{1}{2} \ln(\frac{1-\varepsilon_t}{\varepsilon_t})$
- Cập nhật strong learner: $H_t(x) = H_{t-1}(x) + \alpha_t h_t(x)$
- Cập nhật lại weights:
  - $w_i^{t+1} = w_i^t e^{-y_i \alpha_t h_t(x_i)}, i \in \{1, 2, ..., n\}$
  - Chuẩn hóa lại weights: $w_i^{t+1} \leftarrow \frac{w_i^{t+1}}{\sum_i w_i^{t+1}}$, có nghĩa là $\sum_i w_i^{t+1} = 1$

B3: Ouput chính là dấu của biểu thức tổng các weak-learners nhân với trọng số của chúng, hay

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$$



4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Implement model:

Thư viện cần dùng:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from typing import Optional

from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, recall_score
from sklearn.metrics import classification_report, log_loss

import warnings
warnings.filterwarnings('ignore')
```

4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Implement model:

Xây dựng Model:

Vì ***y*** nhận 2 giá trị -1 và 1, nên mình cần có 1 hàm kiểm tra gía trị của ***y***

```
def check(y):
    assert set(y) == {-1,1}
    return y
```

4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Bảng giải thích các biến

Tên biến	Chiều	Ký hiệu trong giải thuật	Ý nghĩa
sample_weights	(T,n)	**w**	Ma trận trọng số các điểm trên các weak-learners
current_sew	(n,)		Trọng số các điểm tại weak-learners thứ t
new_sew	(n,)		Trọng số các điểm tại weak-learners thứ t+1
stumps	(T,)	**h**	Danh sách các weak-learners)
stump	(1,)		Weak-learners thứ t
stump_weights	(T,)	**A**	Trọng số các weak-learners
stump_weight	(1,)		Trọng số weak-learners thứ t
errors	(T,)	**E**	Danh sách tỉ lệ lỗi của các weak-learners
error	(1,)		Tỉ lệ lỗi của weak-learners thứ t
predict(X)	(n,)		Ouput thuật toán

4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Khởi tạo các biến

```
def init_model(iters, X):
    n = X.shape[0]
    sample_weights = np.zeros((iters, n))
    stumps = np.zeros(iters, dtype= object)
    stump_weights = np.zeros(iters)
    errors = np.zeros(iters)
    return stumps, stump_weights, sample_weights, errors
```

Vì mục tiêu là hiểu AdaBoost, nên weak-learners ht(x)*ht*(*x*) ta sử dụng *DecisionTreeClassifier* được xây dựng trong *sklearn.ensemble* với trọng số (max_depth=1, max_leaf_nodes=2)

4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

```
def AdaBoostClf(X, y, iters= 10):
    n = X.shape[0]
    # Check y
    y = check(y)
    # Initialize
    stumps, stump_weights, sample_weights, errors =
init_model(iters= iters, X= X)

    # First weight = 1/n
    sample_weights[0] = np.ones(shape= n) / n

    for i in range(iters):
        # Fit for stump: weak learner
        current_sew = sample_weights[i]
        stump = DecisionTreeClassifier(max_depth= 1,
max_leaf_nodes= 2)
        stump = stump.fit(X, y, sample_weight= current_sew)

        # Calculate error
        stump_pred = stump.predict(X)
        error = current_sew[stump_pred != y].sum()
        stump_weight = np.log((1 - error) / error) / 2
```

```
# New sample weight
        new_sew = current_sew * np.exp(-1 *
stump_weight * y * stump_pred)

        # Renormalize weights
        new_sew = new_sew / new_sew.sum()

        # If not last iter, update sample weights
for i+1
        if (i + 1) < iters:
            sample_weights[i+1] = new_sew

        # Save result
        errors[i] = error
        stumps[i] = stump
        stump_weights[i] = stump_weight

    return stumps, stump_weights, sample_weights
```

4.6 Kỹ thuật Boosting

4.6.2 AdaBoost

Predictions

Như đã đề cập ở trên, kết quả phân loại từng điểm chính là dấu của biểu thức

$$H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$$

với (+) nghĩa là y(x)=1$y(x)=1$ và (-) là y(x)=−1$y(x)=-1$

```
def predict(X, stumps, stump_weights):
    stump_preds = np.array([stump.predict(X) for stump in stumps])
    return np.sign(np.dot(stump_weights, stump_preds))
```

4.6 Kỹ thuật Boosting

4.6.2 Gradient Boosting

Gradient Boosting là một dạng tổng quát hóa của AdaBoost. Cụ thể như sau, vẫn vấn đề tối ưu ban đầu

$$\boldsymbol{min_{c_n,w_n} L(y, W_{n-1} + c_n w_n)}$$

coi chuỗi các model boosting là một hàm số W*W*, thì mỗi hàm learner có thể coi là một tham số w*w*. Đến đây, để cực tiểu hóa hàm loss L(y,W), chúng ta áp dụng Gradient Descent

$$W_n = W_{n-1} - \eta \frac{\partial}{\partial w} L(W_{n-1})$$

Đến đây, ta có thể thấy mối quan hệ liên quan sau

$$c_n w_n \approx -\eta \frac{\partial}{\partial w} L(W_{n-1})$$

với w_n là model được thêm vào tiếp theo. Khi đó, model mới cần học để fit để vào giá trị $-\eta \frac{\partial}{\partial w} L(W_{n-1})$. (Giá trị $-\eta \frac{\partial}{\partial w} L(W_{n-1})$ còn có 1 tên gọi khác là **pseudo-residuals**)

4.6 Kỹ thuật Boosting

4.6.2 Gradient Boosting

Quá trình triển khai thuật toán như sau:

•Khởi tạo giá trị pseudo-residuals là bằng nhau cho từng điểm dữ liệu

•Tại vòng lặp thứ i

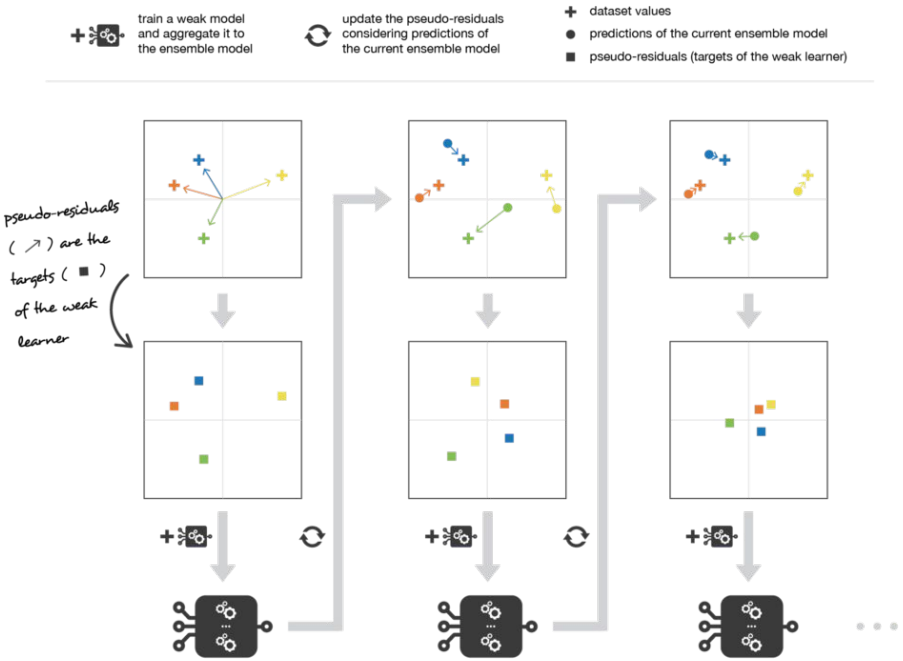
- Train model mới được thêm vào để fit vào giá trị của pseudo-residuals đã có
- Tính toán giá trị confidence score ci*ci* của model vừa train
- Cập nhật model chính W=W+ci∗wi $W = W + c_i * w_i$
- Cuối cùng, tính toán giá trị pseudo-residuals $-\eta \frac{\partial}{\partial w} L(W_{n-1})$ để làm label cho model tiếp theo

•Sau đó lặp lại với vòng lặp i + 1.

4.6 Kỹ thuật Boosting

4.6.2 Gradient Boosting

Phương pháp cập nhật lại trọng số của điểm dữ liệu của AdaBoost cũng là 1 trong các case của Gradient Boosting. Do đó, Gradient Boosting bao quát được nhiều trường hợp hơn.



4.6 Kỹ thuật Boosting

4.6.2 Gradient Boosting

```python
# Import các thư viện cần thiết
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification

# 1. Tạo dữ liệu mẫu
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, random_state=42)

# 2. Chia dữ liệu thành tập huấn luyện và kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# 3. Khởi tạo và huấn luyện mô hình Gradient Boosting
gb_model = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=42)
gb_model.fit(X_train, y_train)
```

```python
# 4. Dự đoán và đánh giá mô hình
y_pred = gb_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Độ chính xác: {accuracy:.2f}")

# 5. (Tùy chọn) Điều chỉnh tham số với
GridSearchCV
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [50, 100],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 5]
}
grid_search =
GridSearchCV(GradientBoostingClassifier(),
param_grid, cv=5)
grid_search.fit(X_train, y_train)
print(f"Tham số tốt nhất:
{grid_search.best_params_}")
```

4.6 Kỹ thuật Boosting

4.6.3 XGBoost (Extreme Gradient Boosting)

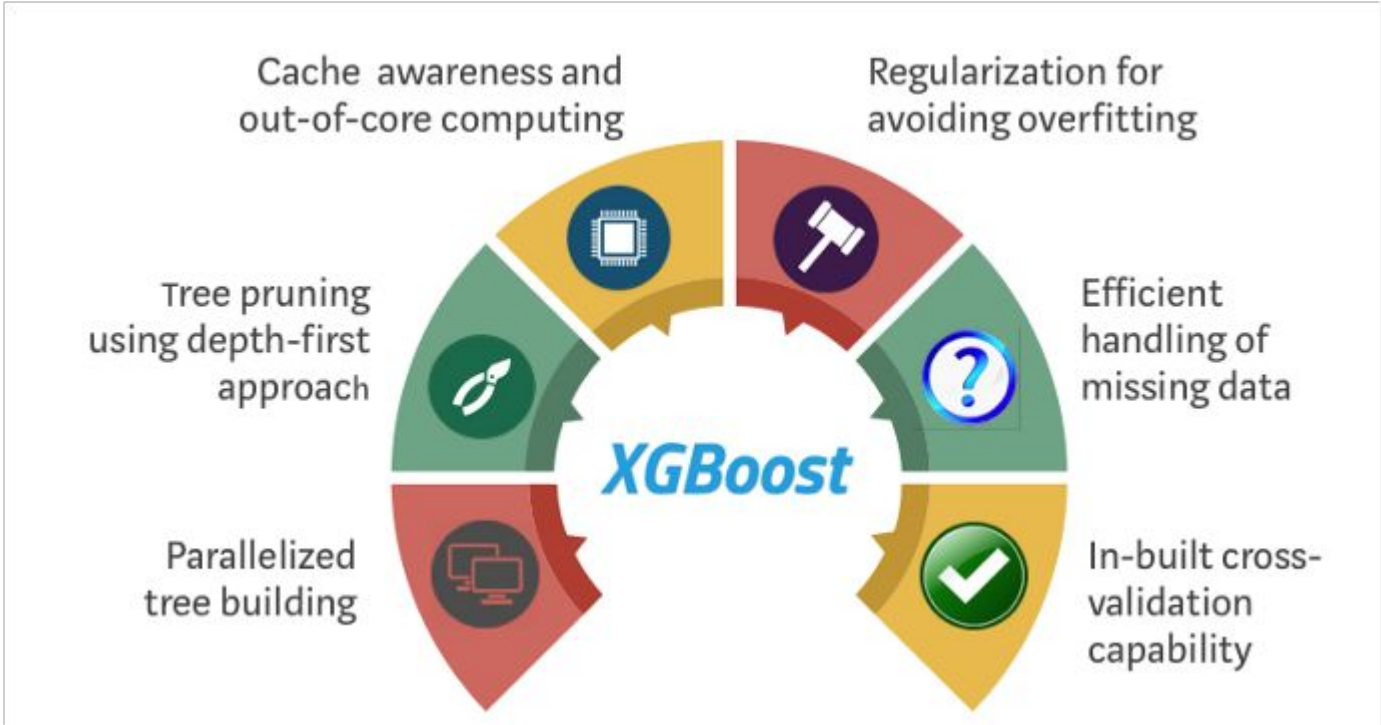
- XGBoost (Extreme Gradient Boosting) là một giải thuật được base trên gradient boosting, tuy nhiên kèm theo đó là những cải tiến to lớn về mặt tối ưu thuật toán, về sự kết hợp hoàn hảo giữa sức mạnh phần mềm và phần cứng, giúp đạt được những kết quả vượt trội cả về thời gian training cũng như bộ nhớ sử dụng.

- Mã nguồn mở với ~350 contributors và ~3,600 commits trên Gihub, XGBoost cho thấy những khả năng ứng dụng đáng kinh ngạc của mình như :

- XGBoost có thể được sử dụng để giải quyết được tất cả các vấn đề từ hồi quy (regression), phân loại (classification), ranking và giải quyết các vấn đề do người dùng tự định nghĩa.
- XGBoost hỗ trợ trên Windows, Linux và OS X.
- Hỗ trợ tất cả các ngôn ngữ lập trình chính bao gồm C ++, Python, R, Java, Scala và Julia.
- Hỗ trợ các cụm AWS, Azure và Yarn và hoạt động tốt với Flink, Spark và các hệ sinh thái khác.
- ...

4.6 Kỹ thuật Boosting

4.6.3 XGBoost (Extreme Gradient Boosting)



4.6 Kỹ thuật Boosting

4.6.3 XGBoost (Extreme Gradient Boosting)

```python
# Import các thư viện cần thiết
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification

# 1. Tạo dữ liệu mẫu
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, random_state=42)

# 2. Chia dữ liệu thành tập huấn luyện và kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# 3. Khởi tạo và huấn luyện mô hình Gradient Boosting
gb_model = XGBClassifier(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=42)
gb_model.fit(X_train, y_train)
```

```python
# 4. Dự đoán và đánh giá mô hình
y_pred = gb_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Độ chính xác: {accuracy:.2f}")

# 5. (Tùy chọn) Điều chỉnh tham số với
GridSearchCV
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [50, 100],
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 5]
}
grid_search =
GridSearchCV(GradientBoostingClassifier(),
param_grid, cv=5)
grid_search.fit(X_train, y_train)
print(f"Tham số tốt nhất:
{grid_search.best_params_}")
```

4.6 Kỹ thuật Boosting

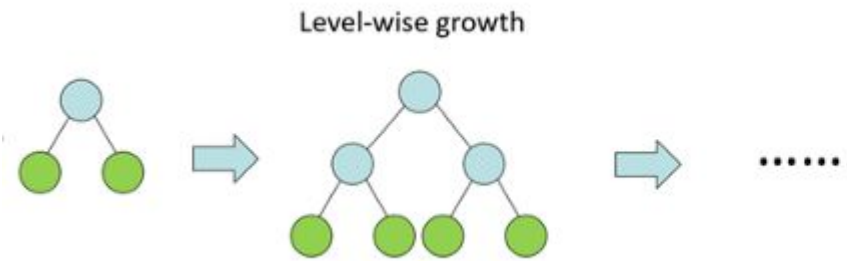
4.6.4 LightGBM

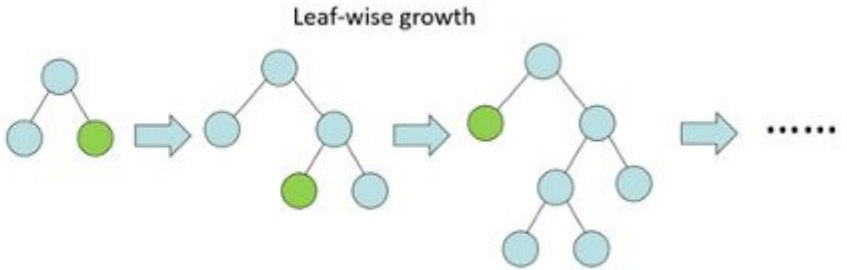
- **XGBoost** cho kết quả tốt nhưng training chậm trên tập dữ liệu lớn.
- **LightGBM** được ưa chuộng hơn nhờ tốc độ nhanh và tối ưu bộ nhớ.
- **LightGBM** dùng histogram-based algorithms thay cho pre-sort-based, giúp:
- Tăng tốc training
- Giảm bộ nhớ sử dụng
- **LightGBM** còn vượt trội nhờ 2 thuật toán:
- **GOSS** (Gradient-based One-Side Sampling) – giảm số lượng mẫu cần tính gradient.
- **EFB** (Exclusive Feature Bundling) – gom nhóm các đặc trưng rời rạc không giao nhau.
- **LightGBM** xây cây theo leaf-wise, tối ưu toàn cây → thường hiệu quả hơn so với:
- **Level-wise** (dùng trong XGBoost) – xây cây đều theo từng tầng.

4.6 Kỹ thuật Boosting

4.6.4 LightGBM

Note: *Leaf-wise tuy tốt, nhưng với những bộ dữ liệu nhỏ, các tree xây dựng dựa trên leaf-wise thường dẫn đến* ***overfit khá sớm****. Do đó, lightgbm sử dụng thêm 1 hyperparameter là maxdepth nhằm cố gắng hạn chế điều này. Dù vậy, LightGBM vẫn được khuyến khích sử dụng khi bộ dữ liệu là đủ to.*





4.6 Kỹ thuật Boosting

4.6.3 XGBoost (Extreme Gradient Boosting)

```
# Import các thư viện cần thiết
import lightgbm as lgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification

# 1. Tạo dữ liệu mẫu
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=15, random_state=42)

# 2. Chia dữ liệu thành tập huấn luyện và kiểm tra
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# 3. Tạo dataset cho LightGBM
train_data = lgb.Dataset(X_train, label=y_train)
test_data = lgb.Dataset(X_test, label=y_test,
reference=train_data)
```

```
# 4. Thiết lập tham số và huấn luyện mô hình
params = {
    'objective': 'binary',  # Phân loại nhị phân
    'metric': 'binary_logloss',
    'num_leaves': 31,
    'learning_rate': 0.1,
    'max_depth': 3,
    'random_state': 42
}
lgb_model = lgb.train(params, train_data,
num_boost_round=100, valid_sets=[test_data])

# 5. Dự đoán và đánh giá
y_pred = lgb_model.predict(X_test)
y_pred = [1 if pred > 0.5 else 0 for pred in
y_pred]  # Chuyển xác suất thành nhãn
accuracy = accuracy_score(y_test, y_pred)
print(f"Độ chính xác: {accuracy:.2f}")
```