**Support Vector Machine**

1. **Giới thiệu**

* Máy vectơ hỗ trợ (Support vector machine - SVM) được đề cử bởi V. Vapnik và các đồng nghiệp của ông vào những năm 1970s ở Nga, và sau đó đã trở nên nổi tiếng và phổ biến vào những năm 1990s
* SVM là một phương pháp phân lớp tuyến tính (linear classifier), với mục đích xác định một siêu phẳng (hyperplane) để phân tách hai lớp của dữ liệu – ví dụ: lớp các ví dụ có nhãn dương (positive) và lớp các ví dụ có nhãn âm (negative)
* Các hàm nhân (kernel functions), cũng được gọi là các hàm biến đổi (transformation functions), được dùng cho các trường hợp phân lớp phi tuyến.
* SVM có một nền tảng lý thuyết chặt chẽ – dựa trên nhiều định lý toán học
* SVM là một phương pháp tốt (phù hợp) đối với những bài toán phân lớp có không gian biểu diễn thuộc tính lớn– các đối tượng cần phân lớp được biểu diễn bởi một tập rất lớn các thuộc tính
* SVM đã được biết đến là một trong số các phương pháp phân lớp tốt nhất đối với các bài toán phân lớp văn bản (text/document classification)
* *Ví dụ:*

Biểu diễn tập r các ví dụ huấn luyện (training examples) :

{(

* là một vectơ đầu vào được biểu diễn trong không gian

X ⊆ Rn

* yi là một nhãn lớp (giá trị đầu ra), yi ∈ {1,−1}
  + yi = 1: lớp dương (positive);
  + yi = −1: lớp âm (negative)

Đối với một ví dụ :

SVM xác định một hàm phân tách tuyến tính:

* + w là vectơ trọng số các thuộc tính
  + b là một giá trị số thực

1. **Một số khái niệm**
   1. Khoảng cách từ một điểm đến một siêu mặt phẳng

Trong không gian 2 chiều, ta biết rằng khoảng cách từ một điểm có toạ độ tới đường thẳng có phương trình được xác định bởi:

A math equations with black text

Description automatically generated with medium confidence

Trong không gian 3 chiều, ta biết rằng khoảng cách từ một điểm có toạ độ tới đường thẳng có phương trình được xác định bởi:

A black and white math equation

Description automatically generated with medium confidence

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

Khoảng cách từ một điểm (vector) có toạ độ tới *siêu mặt phẳng* (*hyperplane*) có phương trình được xác định bởi:

A black text on a white background

Description automatically generated

Với với d là số chiều của không gian.

* 1. Mặt siêu phẳng phân cách
     + Mặt siêu phẳng phân tách các ví dụ huấn luyện lớp dương và các ví dụ huấn luyện lớp âm:

A diagram of a mathematical equation

Description automatically generated

* + - Còn được gọi là ranh giới (bề mặt) quyết định
    - Tồn tại nhiều mặt siêu phẳng phân tách

A diagram of a graph

Description automatically generated with medium confidence

* 1. Mặt siêu phẳng có lề cực đại
* SVM lựa chọn mặt siêu phẳng phân tách có lề (margin) lớn nhất

A diagram of mathematical equations

Description automatically generated

(1)

* *Lý thuyết học máy đã chỉ ra rằng một mặt siêu phẳng phân tách như thế sẽ tối thiểu hóa giới hạn lỗi (phân lớp) mắc phải*

1. **Hard Margin SVM**
   1. SVM – Dữ liệu phân tách được tuyến tính

* Giả sử rằng tập dữ liệu (tập các ví dụ huấn luyện) có thể phân tách được một cách tuyến tính.
* Xét một ví dụ của lớp dương (1) và một ví dụ của lớp âm (,-1) gần nhất đối với siêu phẳng phân tách
* Định nghĩa 2 siêu phẳng lề song song với nhau

* + đi qua và song song với

* + đi qua  và song song với

Sao cho:

* + - nếu
    - nếu
  1. Tính toán mức lề
* Mức lề (margin) là khoảng cách giữa 2 siêu phẳng lề H+ và H-. Trong hình vẽ (1) :
* là khoảng cách từ và
* là khoảng cách từ và
* ( là mức lề
* Theo lý thuyết đại số vectơ, khoảng cách (trực giao) từ một điểm đến mặt siêu phẳng là :

A math equation with a plus and b

Description automatically generated

Trong đó ||w|| là độ dài của w :



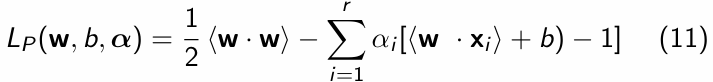
* Tính toán :
  + Áp dụng Eq.3-4: A black and white math equation

    Description automatically generated with medium confidence
* Tính toán :
  + Áp dụng Eq.3-4: 
* Tính toán mức lề (:

A math equation with numbers and symbols

Description automatically generated

* 1. Giải bài toán cực tiểu hóa có ràng buộc
* Biểu thức Lagrange :



Trong đó (là **các hệ số nhân Lagrange**

* Lý thuyết tối ưu chỉ ra rằng một lời giải tối ưu cho (11) phải thỏa mãn các điều kiện nhất định, được gọi là các **điều kiện Karush-Kuhn-Tucker** là các điều kiện cần (nhưng không phải là các điều kiện đủ)
* Tập điều kiện Karush-Kuhn-Tucker :

A math equations with numbers and symbols

Description automatically generated

* + (14) chính là tập các ràng buộc ban đầu
  + Điều kiện bổ sung (16) chỉ ra rằng chỉ những ví dụ (điểm dữ liệu) thuộc các mặt siêu phẳng lề (mới có >0 - bởi vì với những ví đụ đó thì 
* Những ví dụ (điểm dữ liệu) này được gọi là **các vectơ hỗ trợ!**
  + Đối với các ví dụ khác (không phải là các vectơ hỗ trợ) thì =0
* *Trong trường hợp tổng quát, các điều kiện Karush-KuhnTucker là cần đối với một lời giải tối ưu, nhưng chưa đủ*
* *Tuy nhiên, đối với bài toán cực tiểu hóa đang xét có hàm mục tiêu lồi (convex) và các ràng buộc tuyến tính, thì các điều kiện Karush-Kuhn-Tucker là cần và đủ đối với một lời giải tối ưu*
* **Giải quyết bài toán tối ưu này vẫn là một nhiệm vụ khó khăn – do sự tồn tại của các ràng buộc bất đẳng thức!**
* Phương pháp Lagrange giải quyết bài toán tối ưu hàm lồi dẫn đến một biểu thức **đối ngẫu (dual)** của bài toán tối ưu
  1. Biểu thức đối ngẫu
* Để thu được biểu thức đối ngẫu từ biểu thức ban đầu:
* Gán giá trị bằng 0 đối với các đạo hàm bộ phận của biểu thức Lagrange trong (11) đối với **các biến ban đầu** (**w** và b)
* Sau đó, áp dụng các quan hệ thu được đối với biểu thức Lagrange
  + Áp dụng các biểu thức (12-13) vào biểu thức Lagrange ban đầu (11) để loại bỏ **các biến ban đầu** (**w** và b)
* Biểu thức đối ngẫu :

A black and white math symbol

Description automatically generated

* Cả hai biểu thức (11) và (17) đều là các biểu thức Lagrange
  + Dựa trên cùng một hàm mục tiêu– nhưng với các ràng buộc khác nhau
  + Lời giải tìm được, bằng **cách cực tiểu hóa hoặc cực đại hóa**
* **Bài toán đối ngẫu:**

**A number and mathematical symbols

Description automatically generated with medium confidenceCực đại hóa:  (18)**

**Với điều kiện**

* Giải quyết biểu thức (18), ta thu được các hệ số nhân Lagrange (các hệ số này sẽ được dùng để tính w và b)
  1. Tính toán các giá trị  **và**
* Gọi SV là tập các vectơ hỗ trợ
  + SV là tập con của tập r các ví dụ huấn luyện ban đầu
    - với các vecto hỗ trợ
    - với các vecto không phải vecto hỗ trợ
* Sử dụng biểu thức (12), ta có thể tính được giá trị **:**

A black text with black letters

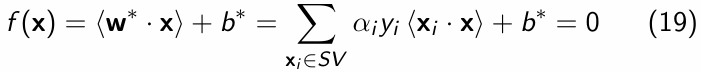
Description automatically generated

* Sử dụng biểu thức (16) và (bất kỳ) một vectơ hỗ trợ, ta có:



Vì với các vecto hỗ trợ

* 
* 
  1. Ranh giới quyết định phân lớp
  + **Ranh giới quyết định phân lớp** được xác định bởi siêu phẳng:



* Đối với một ví dụ cần phân lớp **z**, cần tính giá trị:



* **Nếu biểu thức (20) trả về giá trị 1, thì ví dụ z được phân vào lớp có nhãn dương (positive); ngược lại, được phân vào lớp có nhãn âm (negative)**
* *Việc phân lớp này:* 
  + *Chỉ phụ thuộc vào các vectơ hỗ trợ*
  + *Chỉ cần giá trị tích vô hướng (tích trong) của 2 vectơ (chứ không cần biết giá trị của 2 vectơ đấy)*
  1. Tóm tắt
* Với bài toán binary classification mà 2 classes là linearly separable, có vô số các siêu mặt phẳng giúp phân biệt hai classes, tức mặt phân cách. Với mỗi mặt phân cách, ta có một classifier. Khoảng cách gần nhất từ 1 điểm dữ liệu tới mặt phân cách ấy được gọi là margin của classifier đó.
* Support Vector Machine là bài toán đi tìm mặt phân cách sao cho margin tìm được là lớn nhất, đồng nghĩa với việc các điểm dữ liệu an toàn nhất so với mặt phân cách.
* Bài toán tối ưu trong SVM là một bài toán lồi với hàm mục tiêu là stricly convex, nghiệm của bài toán này là duy nhất. Hơn nữa, bài toán tối ưu đó là một Quadratic Programming (QP).
* Mặc dù có thể trực tiếp giải SVM qua bài toán tối ưu gốc này, thông thường người ta thường giải bài toán đối ngẫu. Bài toán đối ngẫu cũng là một QP nhưng nghiệm là sparse nên có những phương pháp giải hiệu quả hơn.

1. **Soft Margin SVM**
   1. Linear SVM: Không phân tách được

* Phương pháp SVM trong trường hợp các ví dụ phân lớp tuyến tính nhưng không thể phân tách được:
  + Trường hợp phân lớp tuyến tính và phân tách được là lý tưởng (ít xảy ra)
  + Tập dữ liệu có thể chứa nhiễu, lỗi
* Hai ví dụ nhiều vàđược gán nhãn lớp sai

A diagram of mathematical equations

Description automatically generated

* 1. Nới lỏng các điều kiện
  + Để làm việc với các dữ liệu chứa nhiễu, cần nới lỏng các điều kiện lề (margin constraints) bằng cách sử dụng các biến **slack**  **0** 
    - nếu
    - nếu
  + Với một ví dụ nhiễu/lỗi :
  + là giới hạn trên của lỗi của các ví dụ huấn luyện
  + Các điều kiện mới đối với trường hợp (phân lớp tuyến tính) không thể phân tách được:

A math equations on a white background

Description automatically generated

* **Tích hợp lỗi vào hàm mục tiêu**
  + Bằng cách gán giá trị chi phí (cost) cho các lỗi, và tích hợp chi phí này trong hàm mục tiêu mới:

Cực tiểu hóa : 

Trong đó C(>0) là tham số xác định **mức độ chi phí (penalty degree)** đối với các lỗi

* C càng lớn, mức độ chi phí càng cao với các lỗi
* k=1 thường đc sử dụng
  1. Bài toán tối ưu mới:

Cực tiểu hóa: A black symbols on a white background

Description automatically generated

Với điều kiện: A number and symbol on a white background

Description automatically generated

* Soft-margin SVM
* Biểu thức tối ưu Lagrange :

A black and white math equation

Description automatically generated with medium confidence

Trong đó ((là **các hệ số nhân Lagrange**

* Tập điều kiện Karush-Kuhn-Tucker

A math equations with numbers

Description automatically generated with medium confidence

A math equations with numbers and symbols

Description automatically generated

* 1. Biểu thức đối ngẫu
* Chuyển biểu thức Lagrange từ dạng ban đầu (primal formulation) về dạng đối ngẫu (dual formulation)
  + Gán giá trị bằng 0 cho các đạo hàm bộ phận của biểu thức Lagrange (22) đối với các biến ban đầu (w, b, **)**
  + Thay thế các kết quả thu được vào biểu thức Lagrange ban đầu

→ Sử dụng các kết quả của các biểu thức (23-25) để thay thế vào trong biểu thức Lagrange ban đầu (22)

* Từ biểu thức (25), ta có 

Vì ( nên suy ra điều kiện

* **Biểu thức đối ngẫu**

Cực đại hóa : A black and white text

Description automatically generated

Với điều kiện : A number of numbers and symbols

Description automatically generated with medium confidence

Ta thấy **và các hệ số nhân Lagrange của chúng**  không xuất hiện trong biểu thức đối ngẫu

* Hàm mục tiêu giống hệt như đối với bài toán phân lớp tuyến tính phân tách được

(Linear SVM)

* Khác biệt duy nhất là tập các ràng buộc mới
* **Bài toán đối ngẫu (32) được giải quyết bằng các phương pháp số học, ta được các giá trị (hệ số nhân Lagrange)** 
  1. Tính toán các giá trị  **và**
* **Các giá trị (hệ số nhân Lagrange) αi lời giải được sử dụng để tính toán và** 
  + **Chưa biết !**
* Để tính được
  + Từ (25) và (31), ta suy ra được: =0 nếu
  + Vì vậy, ta có thể sử dụng một ví dụ học thỏa mãn điều kiện (0 < < C) ( là một vecto hỗ trợ!) và (30) với : =0 để tính
* Việc tính toán b\* tương tự như với trường hợp phân lớp tuyến tính phân tách được!
  1. Các đặc điểm quan trọng
* Từ các biểu thức (25-31), ta có thể suy ra các kết luận sau:

A group of math symbols

Description automatically generated

* Biểu thức (33) thể hiện một đặc điểm rất quan trọng của SVM
  + Lời giải được xác định dựa trên rất ít (sparse) các giá trị
    - Rất nhiều ví dụ học nằm ngoài khoảng lề (margin area), và chúng có giá trị bằng 0
    - Các ví dụ **nằm trên lề**  *-* là các vecto hỗ trợ, thì có giá trị khác không (0 < < C)
    - Các ví dụ **nằm trong khoảng lề**  *-* là các ví dụ lỗi/nhiễu thì có giá trị khác không (=C)
  + Nếu không có đặc điểm thưa thớt (sparsity) này, thì phương pháp SVM không thể hiệu quả đối với các tập dữ liệu lớn
  1. Ranh giới quyết định phân lớp
* Ranh giới quyết định phân lớp chính là siêu phẳng:

A black and white math symbol

Description automatically generated

* + Rất nhiều ví dụ học có giá trị = 0 (chính là đặc điểm thưa thớt– sparsity– của phương pháp SVM)
* Đối với một ví dụ cần phân loại **z**, nó được phân loại bởi: 
* Cần xác định giá trị phù hợp của tham số C (trong hàm tối ưu mục tiêu)
  + Thường được xác định bằng cách sử dụng một tập tối ưu (validation set)
  1. Tóm tắt
* SVM thuần (Hard Margin SVM) hoạt động không hiệu quả khi có nhiễu ở gần biên hoặc thậm chí khi dữ liệu giữa hai lớp gần linearly separable. Soft Margin SVM có thể giúp khắc phục điểm này.
* Trong Soft Margin SVM, chúng ta chấp nhận lỗi xảy ra ở một vài điểm dữ liệu. Lỗi này được xác định bằng khoảng cách từ điểm đó tới đường biên tương ứng. Bài toán tối ưu sẽ tối thiểu lỗi này bằng cách sử dụng thêm các biến được gọi là slack varaibles.
* Để giải bài toán tối ưu, có hai cách khác nhau. Mỗi cách có những ưu, nhược điểm riêng, các bạn sẽ thấy trong các bài tới.
* Cách thứ nhất là giải bài toán đối ngẫu. Bài toán đối ngẫu của Soft Margin SVM rất giống với bài toán đối ngẫu của Hard Margin SVM, chỉ khác ở ràng buộc chặn trên của các nhân tử Laggrange. Ràng buộc này còn được gọi là box costraint.
* Cách thứ hai là đưa bài toán về dạng không ràng buộc dựa trên một hàm mới gọi là hinge loss. Với cách này, hàm mất mát thu được là một hàm lồi và có thể giải được khá dễ dàng và hiệu quả bằng các phương pháp Gradient Descent.
* Trong Soft Margin SVM, có một hằng số phải được chọn, đó là**C**. Hướng tiếp cận này còn được gọi là C-SVM. Ngoài ra, còn có một hướng tiếp cận khác cũng hay được sử dụng, gọi là ν-SVM, bạn đọc có thể đọc thêm [tại đây](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.2928&rep=rep1&type=pdf).

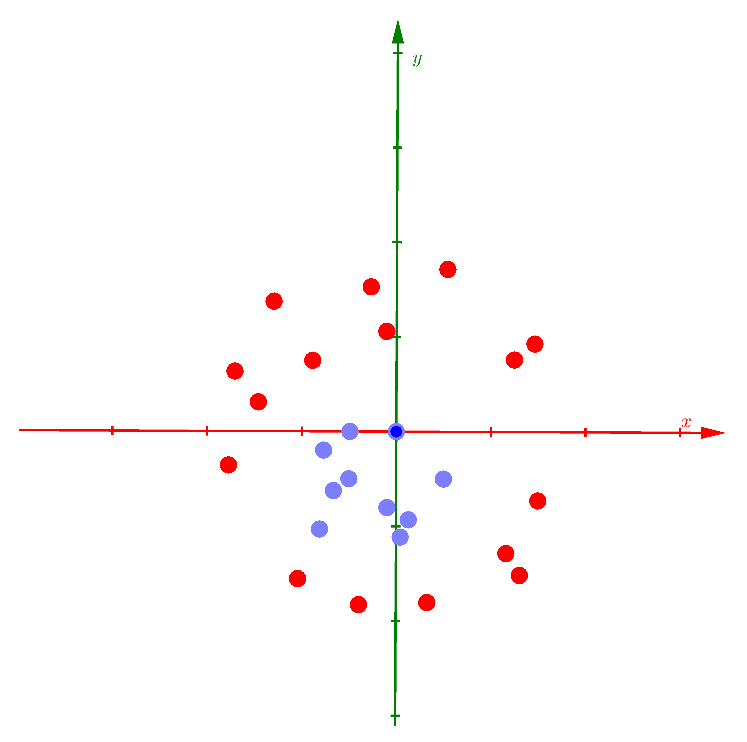
1. **Kernel SVM**
   1. Giới thiệu

Kernel SVM, tức việc áp dụng SVM lên bài toán mà dữ liệu giữa hai classes là hoàn toàn *không linear separable (không phân biệt tuyến tính)*

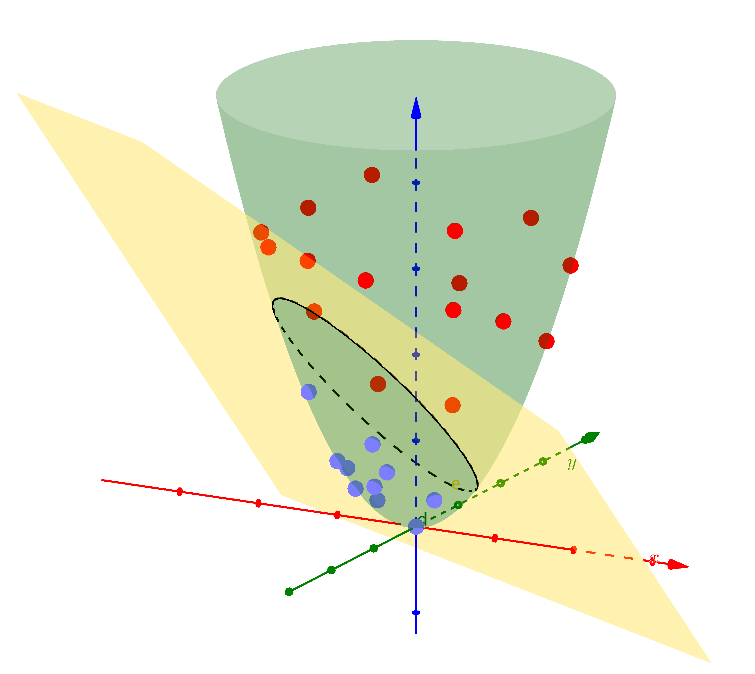
Ý tưởng cơ bản của Kernel SVM và các phương pháp kernel nói chung là tìm một phép biến đổi sao cho dữ liệu ban đầu là *không phân biệt tuyến tính* được biến sang không gian mới. Ở không gian mới này, dữ liệu trở nên *phân biệt tuyến tính*.

Ví dụ:

Dưới đây với việc biến dữ liệu *không phân biệt tuyến tính* trong không gian hai chiều thành *phân biệt tuyến tính* trong không gian ba chiều bằng cách giới thiệu thêm một chiều mới



1. Dữ liệu của hai classes là *không phân biệt tuyến tính* trong không gian hai chiều.



1. Nếu coi thêm chiều thứ ba là một hàm số của hai chiều còn lại  các điểm dữ liệu sẽ được phân bố trên 1 parabolic và đã trở nên *phân biệt tuyến tính*. Mặt phẳng màu vàng là mặt phân chia, có thể tìm được bởi Hard/Soft Margin SVM.

A diagram of a graph

Description automatically generated

1. Giao điểm của mặt phẳng tìm được và mặt parabolic là một đường ellipse, khi chiếu toàn bộ dữ liệu cũng như đường ellipse này xuống không gian hai chiều ban đầu, ta đã tìm được đường phân chia hai classses.

* Kernel SVM là việc đi tìm một hàm số biến đổi dữ liệu**x** từ không gian *feature* ban đầu thành dữ liệu trong một không gian mới bằng hàm số **Φ(x)** *(Hàm số này cần thỏa mãn mục đích của chúng ta: trong không gian mới, dữ liệu giữa hai classes là phân biệt tuyến tính hoặc gần như phần biệt tuyến tính.)*
* Các hàm **Φ()** thường tạo ra dữ liệu mới có số chiều cao hơn số chiều của dữ liệu ban đầu, thậm chí là vô hạn chiều.
  1. Cơ sở toán học
* Bài toán đối ngẫu trong Soft Margin SVM cho dữ liệu *gần phân biệt tuyến tính (có nhiễu/lỗi)*:

A math equations and formulas

Description automatically generated with medium confidence

Trong đó:

A screenshot of a computer

Description automatically generated

Sau khi giải được λ cho bài toán (1), *nhãn* của một điểm dữ liệu mới sẽ được xác định bởi dấu của biểu thức:

A mathematical equation with numbers

Description automatically generated with medium confidence

Trong đó:

A white background with black text

Description automatically generated

*Với dữ liệu thực tế, rất khó có dữ liệu gần như phân biệt tuyến tính, nghiệm của (1) có thể không thực sự tạo ra một bộ phân lớp tốt.*

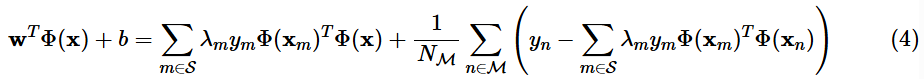
* Giả sử rằng ta có thể tìm được hàm số **Φ()** sao cho sau khi được biến đổi sang không gian mới, mỗi điểm dữ liệu xx trở thành **Φ(x)**, và trong không gian mới này, dữ liệu trở nên *gần phân biệt tuyến tính*.

Trong không gian mới, (1) có dạng:

A black and white math equations

Description automatically generated with medium confidence

Và nhãn của một điểm dữ liệu mới được xác định bởi dấu của biểu thức:



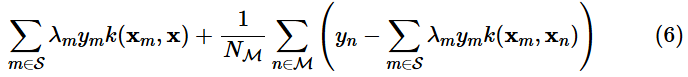
Trong bài toán (3) và biểu thức (4), chúng ta không cần tính trực tiếp **Φ(x)** cho mọi điểm dữ liệu. Chúng ta chỉ cần tính được  dựa trên hai điểm dữ liệu **x, z** bất kì! **(kernel trick)**

* *Những phương pháp dựa trên kỹ thuật này, tức thay vì trực tiếp tính tọa độ của một điểm trong không gian mới, ta đi tính tích vô hướng giữa hai điểm trong không gian mới, được gọi chung là****kernel method****.*
* Bằng cách định nghĩa hàm kernel , ta có thể viết lại (3), (4) như sau
* (3) có dạng:

A math equations on a white background

Description automatically generated

* (4) có dạng:



* Ví dụ

A math equations on a white background

Description automatically generated

* 1. Hàm số kernel (kernel method)
     1. Tính chất
* Các hàm kerrnel cần có các tính chất:
  + Đối xứng: k(x,z)=k(z,x)
  + *Về lý thuyết*, hàm kerrnel cần thỏa mãn [điều kiện Mercer](https://en.wikipedia.org/wiki/Mercer%27s_theorem" \l "Mercer.27s_condition) (đảm bảo hàm mục tiêu của (5) là hàm lồi):
  + A close up of a sign

    Description automatically generated with medium confidence
  + *Trong thực hành, có một vài hàm số***k()***không thỏa mãn điều kiện Merrcer nhưng vẫn cho kết quả chấp nhận được. Những hàm số này vẫn được gọi là kernel method.*
* Nếu một hàm kerrnel thỏa mãn điều kiện (7), xét , ta có:

A black text with black letters

Description automatically generated

Với **K** là một ma trận đối xứng mà phần tử ở hàng thứ nn cột thứ mm của nó được định nghĩa bởi: 

Từ (8) ta suy ra **K** là một ma trận nửa xác định dương.

* Bài toán tối ưu (5) có ràng buộc là lồi và hàm mục tiêu là một hàm lồi (một quadratic form)
  + 1. Một số hàm kernel thông dụng
       1. Linear

Đây là trường hợp đơn giản với kernel chính tích vô hướng của hai vector:



* + - 1. Polynomial



Với **d** là một số dương để chỉ bậc của đa thức. dd có thể không là số tự nhiên vì mục đích chính của ta không phải là bậc của đa thức mà là cách tính kernel.

* *Polynomial kernel có thể dùng để mô tả hầu hết các đa thức có bậc không vượt quá dd nếu***d***là một số tự nhiên.*
  + - 1. Radial Basic Function

Radial Basic Function (RBF) kernel hay Gaussian kernel được sử dụng nhiều nhất trong thực tế, và là lựa chọn mặc định trong sklearn. Nó được định nghĩa bởi:



* + - 1. Sigmoid

[Sigmoid function](https://machinelearningcoban.com/2017/01/27/logisticregression/#sigmoid-function) cũng được sử dụng làm kernel:



* + - 1. Bảng tóm tắt các kernel thông dụng

A white sheet with red text

Description automatically generated

* + - 1. Tự định nghĩa

**Ngoài các hàm kernel thông dụng như trên, chúng ta cũng có thể tự định nghĩa các kernel của mình.**

* 1. Tóm tắt
* Nếu dữ liệu của hai lớp là không phân biệt tuyến tính, chúng ta có thể tìm cách biến đổi dữ liệu sang một không gian mới sao cho trong không gian mới ấy, dữ liệu của hai lớp là phân biệt tuyến tính hoặc gần phân biệt tuyến tính.
* Việc tính toán trực tiếp hàm **Φ()** đôi khi phức tạp và tốn nhiều bộ nhớ. Thay vào đó, ta có thể sử dụng **kernel trick**. Trong cách tiếp cận này, ta chỉ cần tính tích vô hướng của hai vector bất kỳ trong không gian mới: 
* Thông thường, các hàm **k()** thỏa mãn điều kiện Merrcer, và được gọi là kernel. Cách giải bài toán SVM với kernel hoàn toàn giống với cách giải bài toán Soft Margin SVM.
* Có 4 loại kernel thông dụng: linear, poly, rbf, sigmoid. Trong đó, rbf được sử dụng nhiều nhất và là lựa chọn mặc định trong các thư viện SVM.
* Với dữ liệu gần phân biệt tuyến tính, linear và poly kernels cho kết quả tốt hơn.

1. **Multi-class Support Vector Machine**
   1. **Giới thiệu**

* Các phương pháp Support Vector Machine đã đề cập (Hard Margin, Soft Margin, Kernel) đều được xây dựng nhằm giải quyết bài toán [Binary Classification](https://machinelearningcoban.com/2017/02/11/binaryclassifiers/), tức bài toán phân lớp với chỉ hai classes.
* Một cách tự nhiên để mở rộng các mô hình này áp dụng cho các bài toán multi-class classification, tức có nhiều classes dữ liệu khác nhau, là [sử dụng nhiều binary classifiers và các kỹ thuật như one-vs-one hoặc one-vs-rest](https://machinelearningcoban.com/2017/02/11/binaryclassifiers/" \l "-binary-classifiers-cho-multi-class-classification-problems).
* Mô hình end-to-end
  + Softmax Regression
  + Convolutional Neural Networks
  + Sự hiệu quả của Softmax Regression nói riêng và Convolutional Neural Networks nói chung là cả *bộ trích chọn đặc trưng* (feature extractor) và *bộ phân lớp* (classifier) được *huấn luyện* đồng thời.
* Những mô hình như thế này còn được gọi là *end-to-end*
* Bias-trick

Thông thường, với một ma trận hệ số  một đầu vào  và vector bias  , ta có thể tính được đầu ra của layer này là:



* 1. Cơ sở toán học
     1. Xây dựng hàm mất mát cho Multi-class Support Vector Machine
        1. Softmax Regression

A diagram of a network

Description automatically generated

* Qua ma trận hệ số **W**, dữ liệu ban đầu trở thành 
* Lúc này, ứng với mỗi một trong **C** classes, chúng ta nhận được một giá trị tương ứng  ứng với class thứ **i**. Giá trị  này còn được gọi là *score* của dữ liệu **x** ứng với class thứ **i**.
* Ý tưởng chính trong **Softmax Regression** là đi tìm ma trận hệ số **W**, mỗi cột của ma trận này ứng với một class, sao cho *score vector* **z** đạt giá trị lớn nhất tại phần tử tương ứng với class chính xác của nó. Sau khi mô hình đã được *trained*, *nhãn* của một điểm dữ liệu mới được tính là vị trí của thành phần score có giá trị lớn nhất trong *score vector.*
* Để huấn luyện trên tập các cặp (*dữ liệu*, *nhãn*), Softmax Regression sử dụng hàm softmax để đưa *score vector* về dạng phân phối xác suất có các phần tử là dương và có tổng bằng 1. Sau đó dùng hàm cross entropy để *ép* vector xác suất này gần với vector xác suất *thật sự* của dữ liệu - tức one-hot vector mà chỉ có đúng 1 phần tử bằng 1 tại class tương ứng, các phần tử còn lại bằng 0.
  + - 1. Hinge loss tổng quát cho Multi-class SVM (hàm mất mát)
* Với Multi-class SVM, trong khi test, class của một input cũng được xác định bởi thành phần có giá trị lớn nhất trong score vector.
* Multi-class SVM xây dựng hàm mất mát dựa trên định nghĩa của *biên an toàn*, giống như trong Hard/Soft Margin vậy.
* Multi-class SVM *muốn* thành phần ứng với *correct class* của *score vector* lớn hơn các phần tử khác, không những thế, nó còn lớn hơn một đại lượng **Δ>0** gọi là *biên an toàn*. A diagram of a line with text

  Description automatically generated with medium confidence
* Giả sử rằng các thành phần của score vector được đánh số thứ tự từ 1. Các classes cũng được đánh số thứ tự từ 1. Giả sử rằng điểm dữ liệu **x** đang xét thuộc class **y** và score vector của nó là vector **z**.
* Thế thì score của *correct class* là , scores của các classes khác là các , **i≠y.** Xét ví dụ như trong Hình 6 với hai score  trong vùng an toàn và  trong vùng vi phạm.
  + Với mỗi score   trong vùng an toàn, loss bằng 0.
  + Với mỗi score  vượt quá điểm an toàn (điểm x đỏ), *loss* do nó gây ra được tính bằng lượng vượt quá so với điểm x đỏ, đại lượng này có thể tính được là:



Tóm lại, với một score , **j≠y,**  *loss* do nó gây ra có thể được viết gọn thành:



Trong đó: là *cột* thứ**j** của ma trận hệ số **W.**

Như vậy, với một điểm dữ liệu (n=1, 2…, N), tổng cộng loss do nó gây ra là:

A black and white text

Description automatically generated

Trong đó  là scores tương ứng với điểm dữ liệu ;  là *correct class* của điểm dữ liệu đó. A math equations on a white background

Description automatically generated

* + - 1. Regularization

Nếu nghiệm tìm được **w** là *hoàn hảo* tức không có score nào *vi phạm* và biểu thức (2) đạt giá trị 0: 

Điều này có nghĩa là **kW** cũng là một nghiệm của bài toán với k>1 bất kỳ. -> vô số nghiệm

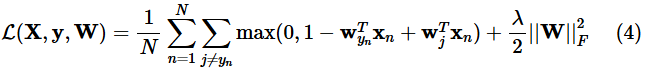
Một phương pháp quen thuộc để tránh hiện tượng này là cộng thêm số hạng [*regularization*](https://machinelearningcoban.com/2017/03/04/overfitting/#-regularization) vào hàm mất mát.

A black text on a white background

Description automatically generated

Trong đó  là [Frobenius norm](https://machinelearningcoban.com/math/" \l "chuan-cua-ma-tran), và **λ** là một giá trị dương giúp cân bằng giữa *data loss* và *regularization loss*, thường được chọn bằng [cross-validation](https://machinelearningcoban.com/2017/03/04/overfitting/#-cross-validation).

* + - 1. Chọn giá trị **Δ**
* Có hai *hyperparameter* trong hàm mất mát (3) là **Δ** và **λ.**
* *Trong thực tế, người ta nhận thấy rằng ΔΔ có thể được chọn bằng 1 mà không ảnh hưởng nhiều tới chất lượng của nghiệm.*
* Hạn chế việc **W**trở nên quá lớn. Việc này đã được điều chỉnh bởi tham số **λ**.
* Cuối cùng, chúng ta sẽ đi tối ưu hàm mất mát sau đây cho Multi-class SVM:



* + - 1. Nhận xét
* ***Soft-margin SVM là một trường hợp đặc biệt của Multi-class SVM***

***A white paper with black text and black text

Description automatically generated***

* + 1. Tính toán hàm mất mát và đạo hàm của nó
* Để tối ưu hàm mất mát, chúng ta sử dụng phương pháp Stochastic Gradient Method.
* Để đảm bảo rằng *loss* và *gradient* được tính một cách chính xác và nhanh, chúng ta sẽ làm từng bước một. Bước thứ nhất là đảm bảo rằng các tính toán là chính xác, dù cách tính có rất chậm. Bước thứ hai là phải đảm bảo có cách tính hiệu quả để thuật toán chạy nhanh hơn. Hai bước này cần được thực hiện trên một lượng dữ liệu nhỏ để đảm bảo chúng được tính chính xác trước khi áp dụng thuật toán vào dữ liệu thật, thường có số điểm dữ liệu lớn và mỗi điểm dữ liệu cũng có số chiều lớn.
  + - 1. Tính hàm mất mát và đạo hàm của nó bằng naïve (đơn giản)
* Dưới đây là cách tính đơn giản *loss* và *gradient* của hàm mất mát trong (4). Chú ý thành phần *regularization*.

*import numpy as np*

*from random import shuffle*

*# naive way to calculate loss and grad*

*def svm\_loss\_naive(W, X, y, reg):*

*d, C = W.shape*

*\_, N = X.shape*

*## naive loss and grad*

*loss = 0*

*dW = np.zeros\_like(W)*

*for n in xrange(N):*

*xn = X[:, n]*

*score = W.T.dot(xn)*

*for j in xrange(C):*

*if j == y[n]:*

*continue*

*margin = 1 - score[y[n]] + score[j]*

*if margin > 0:*

*loss += margin*

*dW[:, j] += xn*

*dW[:, y[n]] -= xn*

*loss /= N*

*loss += 0.5\*reg\*np.sum(W \* W) # regularization*

*dW /= N*

*dW += reg\*W # gradient off regularization*

*return loss, dW*

*# random, small data*

*N, C, d = 10, 3, 5*

*reg = .1*

*W = np.random.randn(d, C)*

*X = np.random.randn(d, N)*

*y = np.random.randint(C, size = N)*

*# sanity check*

*print 'loss without regularization:', svm\_loss\_naive(W, X, y, 0)[0]*

*print 'loss with regularization:'svm\_loss\_naive(W, X, y, .1)[0]*

*loss without regularization: 4.68441457903*

*loss with regularization: 6.25136675351*

* *Cách tính với hai vòng for lồng nhau như trên mô tả lại chính xác loss trong (4) nên sai sót, nếu có, ở đây có thể được kiểm tra và sửa lại dễ dàng.*
* *Việc kiểm tra ở cuối cho cái nhìn ban đầu về hàm mất mát: dương và không có regularization sẽ có loss tổng cộng nhỏ hơn.*

Về cách tính *gradient* cho phần *data loss*, mặc dù [hàm max là](https://machinelearningcoban.com/2017/03/12/convexity/" \l "-cac-tinh-chat-co-ban)*[convex](https://machinelearningcoban.com/2017/03/12/convexity/" \l "-cac-tinh-chat-co-ban)* nhưng nó không có đạo hàm tại mọi nơi. Cụ thể:

A group of numbers and symbols

Description automatically generated with medium confidence

Rõ ràng là các đạo hàm này không xác định tại các điểm mà . Tuy nhiên, khi thực hành, ta có thể giả sử rằng tại 0, các đạo hàm này cũng bằng 0.

Để kiểm tra lại cách tính đạo hàm như trên dựa vào (5) và (6) có chính xác không, chúng ta sẽ làm một bước quen thuộc là so sánh nó với [*numerical gradient*](https://machinelearningcoban.com/2017/01/12/gradientdescent/#kiem-tra-dao-ham). Nếu sự sai khác là nhỏ, nhỏ hơn 1e-7 thì ta có thể coi là *gradient* tính được là chính xác:

*f = lambda W: svm\_loss\_naive(W, X, y, .1)[0]*

*# for checking if calculated grad is correct*

*def numerical\_grad\_general(W, f):*

*eps = 1e-6*

*g = np.zeros\_like(W)*

*# flatening variable -> 1d. Then we need*

*# only one for loop*

*W\_flattened = W.flatten()*

*g\_flattened = np.zeros\_like(W\_flattened)*

*for i in xrange(W.size):*

*W\_p = W\_flattened.copy()*

*W\_n = W\_flattened.copy()*

*W\_p[i] += eps*

*W\_n[i] -= eps*

*# back to shape of W*

*W\_p = W\_p.reshape(W.shape)*

*W\_n = W\_n.reshape(W.shape)*

*g\_flattened[i] = (f(W\_p) - f(W\_n))/(2\*eps)*

*# convert back to original shape*

*return g\_flattened.reshape(W.shape)*

*# compare two ways of computing gradient*

*g1 = svm\_loss\_naive(W, X, y, .1)[1]*

*g2 = numerical\_grad\_general(W, f)*

*print 'gradient difference: %f' %np.linalg.norm(g1 - g2)*

*# this should be very small*

*gradient difference: 0.000000*

* **Sự sai khác là xấp xỉ 0, vậy chúng ta có thể yên tâm khi nói rằng cách tính *gradient* đã thỏa mãn sự *chính xác*, chúng ta cần tính nó một cách *hiệu quả* nữa.**
  + - 1. Tính hàm mất mát và đạo hàm của nó bằng vectorized

A diagram of a mathematical equation

Description automatically generated

* Ở đây, chúng ta tạm quên phần *regularization loss* đi vì cả *loss* và *gradient* của phần này đều có cách tính đơn giản. Với phần *data loss*, chúng ta cũng bỏ qua hệ số  đi cho dễ hình dung.
* Giả sử rằng có 4 classes và mini-batch gồm có 3 điểm dữ liệu ,. 3 điểm này lần lượt thuộc vào các class 1, 3, 2. Các ô có nền màu đỏ nhạt ở mỗi cột tương ứng với *correct class* của điểm dữ liệu của cột đó. Các bước tính *loss* và *gradient* có thể được hình dung như sau:
  + Bước 1: Tính score matrix
  + Bước 2: Với mỗi ô, tính Chú ý rằng ta không cần tính các ô có nền màu đỏ nhạt vì có thể coi chúng bằng 0 do trong biểu thức data loss. Sau khi tính được giá trị của từng ô, ta chỉ quan tâm tới các ô có giá trị lớn hơn 0 - là các ô được tô nền màu xanh lục. Lấy tổng của tất cả các phần tử ở các ô xanh lục, ta sẽ được data loss.
  + Bước 3: Theo công thức (6), với ô màu xanh lục ở hàng 2, cột 1, thì đạo hàm theo vector hệ số ,sẽ được cộng thêm một lượng , và đạo hàm theo vector hệ số , sẽ được trừ đi một lượng ,. Tương tự với các ô màu xanh lục còn lại. Với các ô màu đỏ ở hàng 1 cột 1, chúng ta chú ý rằng trong cùng cột 1, có bao nhiêu ô màu xanh lục thì có bấy nhiêu lần đạo hàm của  bị trừ đi một lượng , . Điều này được suy ra từ (5). Từ đó suy ra trong khối ô vuông thứ 3, giá trị của ô màu đỏ sẽ bằng đối số của tổng số lượng các ô màu xanh lục. Vậy nên ô màu đỏ ở hàng 1 cột 1 phải bằng -2.
  + Bước 4: Bây giờ cộng theo các hàng, ta sẽ được đạo hàm theo hệ số của class tương ứng.
* Bây giờ cộng theo các hàng, ta sẽ được đạo hàm theo hệ số của class tương ứng.

*# more efficient way to compute loss and grad*

***def******svm\_loss\_vectorized****(W, X, y, reg):*

*d, C* ***=*** *W.shape*

*\_, N* ***=*** *X.shape*

*loss* ***=*** *0*

*dW* ***=*** *np.zeros\_like(W)*

*Z* ***=*** *W.T.dot(X)*

*correct\_class\_score* ***=*** *np.choose(y, Z).reshape(N,1).T*

*margins* ***=*** *np.maximum(0, Z* ***-*** *correct\_class\_score* ***+*** *1)*

*margins[y, np.arange(margins.shape[1])]* ***=*** *0*

*loss* ***=*** *np.sum(margins, axis* ***=*** *(0, 1))*

*loss* ***/=*** *N*

*loss* ***+=*** *0.5* ***\**** *reg* ***\**** *np.sum(W* ***\**** *W)*

*F* ***=*** *(margins* ***>*** *0).astype(int)*

*F[y, np.arange(F.shape[1])]* ***=*** *np.sum(****-****F, axis* ***=*** *0)*

*dW* ***=*** *X.dot(F.T)****/****N* ***+*** *reg****\*****W*

***return*** *loss, dW*

* Sau khi đã viết đoạn code mà chúng ta *cho rằng* đã hiệu quả (không còn vòng for nào) này, chúng ta cần phải kiểm chứng hai điều:
  + - Quy trình 4 bước tôi nêu ở trên có chính xác không. Việc này được kiểm chứng bằng cách so sánh đạo hàm này với đạo hàm nhận được bằng cách tính *naive*.
    - Cách tính thứ hai này liệu có thực sự *hiệu quả*, tức có nhanh hơn cách *naive* nhiều không.

*N, C, d = 49000, 10, 3073*

*reg = .1*

*W = np.random.randn(d, C)*

*X = np.random.randn(d, N)*

*y = np.random.randint(C, size = N)*

*import time*

*t1 = time.time()*

*l1, dW1 = svm\_loss\_naive(W, X, y, reg)*

*t2 = time.time()*

*print 'Naive : run time:', t2 - t1, '(s)'*

*t1 = time.time()*

*l2, dW2 = svm\_loss\_vectorized(W, X, y, reg)*

*t2 = time.time()*

*print 'Vectorized: run time:', t2 - t1, '(s)'*

*print 'loss difference:', np.linalg.norm(l1 - l2)*

*print 'gradient difference:', np.linalg.norm(dW1 - dW2)*

***Naive : run time: 34.326472044 (s)***

***Vectorized: run time: 0.267823934555 (s)***

***loss difference: 3.63797880709e-12***

***gradient difference: 2.70855454684e-14***

Kết quả nhận được cho chúng ta thấy rằng cách tính bằng *vectorized* nhanh hơn rất nhiều (khoảng 120 lần) so với cách tính *naive*. Hơn nữa, sự chênh lệch giữa kết quả của hai cách tính là rất nhỏ, đều nhỏ hơn 1e-10 . Vậy thì chúng ta có thể *yên tâm* sử dụng cách *vectorized* này để cập nhật nghiệm.

* + - 1. Gradient Descent cho Multi-class SVM

*# Mini-batch gradient descent*

*def multiclass\_svm\_GD(X, y, Winit, reg, lr=.1, \*

*batch\_size = 100, num\_iters = 1000, print\_every = 100):*

*W = Winit*

*loss\_history = np.zeros((num\_iters))*

*for it in xrange(num\_iters):*

*# randomly pick a batch of X*

*idx = np.random.choice(X.shape[1], batch\_size)*

*X\_batch = X[:, idx]*

*y\_batch = y[idx]*

*loss\_history[it], dW = \*

*svm\_loss\_vectorized(W, X\_batch, y\_batch, reg)*

*W -= lr\*dW*

*if it % print\_every == 1:*

*print 'it %d/%d, loss = %f' \*

*%(it, num\_iters, loss\_history[it])*

*return W, loss\_history*

*N, C, d = 49000, 10, 3073*

*reg = .1*

*W = np.random.randn(d, C)*

*X = np.random.randn(d, N)*

*y = np.random.randint(C, size = N)*

*W, loss\_history = multiclass\_svm\_GD(X, y, W, reg)*

***it 1/1000, loss = 1802.750975***

***it 101/1000, loss = 251.495825***

***it 201/1000, loss = 62.021015***

***it 301/1000, loss = 45.626031***

***it 401/1000, loss = 38.334262***

***it 501/1000, loss = 43.666638***

***it 601/1000, loss = 45.649841***

***it 701/1000, loss = 35.401936***

***it 801/1000, loss = 36.211475***

***it 901/1000, loss = 41.676211***

* Chúng ta thử visisualize giá trị của loss sau mỗi vòng lặp:

*import matplotlib.pyplot as plt*

*# plot loss as a function of iteration*

*plt.plot(loss\_history)*

*plt.show()*

***A white rectangular object with black text

Description automatically generated***

* Từ *lịch sử loss* này ta thấy rằng giá trị của *loss* sau mỗi vòng lặp có xu hướng giảm và hội tụ, đây chính là điều mà chúng ta mong muốn.
  1. Tóm tắt
* [Giống như Softmax Regression](https://machinelearningcoban.com/2017/02/17/softmax/#-boundary-tao-boi-softmax-regression-la-linear), Multi-class SVM vẫn được coi là một bộ phân lớp tuyến tính vì đường phân chia giữa các class là các đường tuyến tính.
* Kernel SVM cũng hoạt động khá tốt, nhưng việc tính toán ma trận kernel có thể tốn nhiều thời gian và bộ nhớ. Hơn nữa, việc mở rộng nó ra cho bài toán multi-class classification thường không hiệu quả bằng Multi-class SVM. Một ưu điểm nữa của Multi-class SVM là nó có thể được tối ưu bằng (Stochastic) Gradient Descnet, tức là nó phù hợp với các bài toán large-scale. Việc boundary giữa các class là tuyến tính có thể được giải quyết bằng cách kết hợp nó với Deep Neurel Networks.
* Có một cách nữa mở rộng hinge loss cho bài toán multi-class classification là dùng loss:  Đây chính là vi phạm lớn nhất, so với tổng vi pham mà chúng ta sử dụng trong bài này.
* Trên thực tế, [**Multi-class SVM và Softmax Regression có hiệu quả tương đương nhau**](http://cs231n.github.io/linear-classify/#svmvssoftmax). Có thể trong một bài toán cụ thể, phương pháp này tốt hơn phương pháp kia, nhưng điều ngược lại xảy ra trong các bài toán khác.

1. **Tài liệu tham khảo**
2. Giáo trình Học máy – Trường Đại Học Xây Dựng Hà Nội (Đào Việt Cường) – Mục I,II,III,IV
3. [Kernel SVM (Vũ Hữu Tiệp)](https://machinelearningcoban.com/2017/04/22/kernelsmv/#-co-so-toan-hoc) - Mục V A
4. [Multi-class (Vũ Hữu Tiệp)](https://machinelearningcoban.com/2017/04/28/multiclasssmv/) - Mục VI