Lecture 9. Kernel Methods

COMP90051 Statistical Machine Learning

Lecturer: Jean Honorio

THE UNIVERSITY OF MELBOURNE

This lecture

- Dual formulation of the SVM
- Kernelisation
  - Basis expansion on dual formulation of SVMs
  - “Kernel trick”; Fast computation of feature space dot product
- Modular learning
  - Separating “learning module” from feature transformation
  - Representer theorem
- Constructing kernels
  - Overview of popular kernels and their properties
  - Mercer’s theorem
  - Learning on unconventional data types

Lagrangian Duality for the SVM

An equivalent formulation, with important consequences.

Soft-margin SVM recap

- Soft-margin SVM objective:

$$\underset{\boldsymbol{w},b,\boldsymbol{\xi}}{\operatorname{argmin}} \left(\frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}\xi_i \right)$$

$$\text{s.t. } y_i(\boldsymbol{w}'\boldsymbol{x}_i + b) \geq 1 - \xi_i \text{ for } i = 1, \dots, n$$

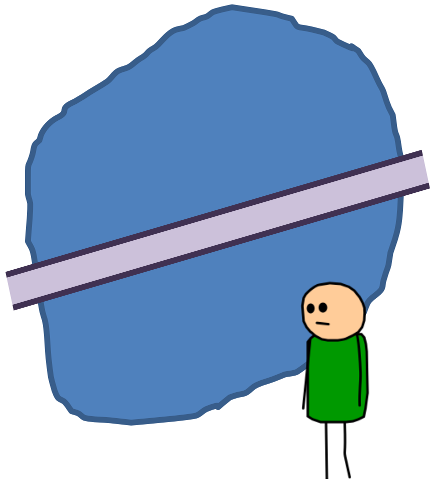
$$\xi_i \geq 0 \text{ for } i = 1, \dots, n$$

- While we can optimise the above "**primal**", often instead work with the **dual**

Constrained optimisation

- Constrained optimisation: canonical form

$$\text{minimise } f(\boldsymbol{x})$$
$$\text{s.t. } g_i(\boldsymbol{x}) \leq 0, \; i = 1, \dots, n$$
$$h_j(\boldsymbol{x}) = 0, \; j = 1, \dots, m$$



  - E.g., find deepest point in the lake, *south of the bridge*

- Gradient descent doesn't immediately apply

- Hard-margin SVM: $\underset{\boldsymbol{w},b}{\operatorname{argmin}} \frac{1}{2}\|\boldsymbol{w}\|^2$ s.t. $1 - y_i(\boldsymbol{w}'\boldsymbol{x}_i + b) \leq 0,$
  $i = 1, \dots, n$

- Method of Lagrange multipliers
  - Transform to unconstrained optimisation
  - Transform primal program to a related dual program, alternate to primal
  - Analyse necessary & sufficient conditions for solutions of both programs

The Lagrangian and duality

- Introduce auxiliary objective function via auxiliary variables

≤0 不等式 等式

$$\mathcal{L}(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\nu}) = f(\boldsymbol{x}) + \sum_{i=1}^{n} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{m} \nu_j h_j(\boldsymbol{x})$$

Primal constraints became penalties

  - Called the *Lagrangian* function
  - New $\boldsymbol{\lambda}$ and $\boldsymbol{\nu}$ are called the *Lagrange multipliers* or *dual variables*

- (Old) primal program: $\min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}\geq \boldsymbol{0},\boldsymbol{\nu}} \mathcal{L}(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\nu})$
- (New) dual program: $\max_{\boldsymbol{\lambda}\geq \boldsymbol{0},\boldsymbol{\nu}} \min_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x},\boldsymbol{\lambda},\boldsymbol{\nu})$

May be easier to solve, advantageous

- Duality theory relates primal/dual:
  - Weak duality: dual optimum ≤ primal optimum
  - For convex programs (inc. SVM!) strong duality: optima coincide!

对 unconstrained problem: derivative is equal to zero

当 derivative =0 时 有限制条件 => 该怎么解

对此问题 有 optimal solution => 且能满足限制条件

Karush-Kuhn-Tucker Necessary Conditions

- Lagrangian: $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = f(\boldsymbol{x}) + \sum_{i=1}^{n} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{m} \nu_j h_j(\boldsymbol{x})$
- Necessary conditions for optimality of a primal solution
- Primal feasibility:
  - $g_i(\boldsymbol{x}^*) \leq 0, i = 1, \ldots, n$
  - $h_j(\boldsymbol{x}^*) = 0, j = 1, \ldots, m$
- Dual feasibility: $\lambda_i^* \geq 0$ for $i = 1, \ldots, n$
- Complementary slackness: $\lambda_i^* g_i(\boldsymbol{x}^*) = 0, i = 1, \ldots, n$
- Stationarity: $\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) = \boldsymbol{0}$ (若没有那些限制，Stationarity 并存在) 求导为0

Souped-up version of necessary condition "derivative is zero" in **unconstrained** optimisation.

在 $L(x,\lambda,\nu)$ 中 $\sum_{i=1}^{n} \lambda_i g_i(x) + \sum_{j=1}^{m} \nu_j h_j(x)$ 都是零.

KKT conditions for hard-margin SVM

The Lagrangian

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + \sum_{i=1}^{n} \lambda_i \left(1 - y_i(\boldsymbol{w}'\boldsymbol{x}_i + b)\right)$$

KKT conditions:

- Primal Feas.: $1 - y_i\left((\boldsymbol{w}^*)'\boldsymbol{x}_i + b^*\right) \leq 0$ for $i = 1, \ldots, n$
- Dual Feas.: $\lambda_i^* \geq 0$ for $i = 1, \ldots, n$
- Comp. slack.: $\lambda_i^* \left(1 - y_i\left((\boldsymbol{w}^*)'\boldsymbol{x}_i + b^*\right)\right) = 0$
- Stationarity: $\nabla_{\boldsymbol{w},b}\mathcal{L}(\boldsymbol{w}^*, b^*, \boldsymbol{\lambda}^*) = \boldsymbol{0}$

Let's minimise Lagrangian w.r.t primal variables

- Lagrangian:

$$\mathcal{L}(\boldsymbol{w}, b, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{w}'\boldsymbol{w} + \sum_{i=1}^{n} \lambda_i - \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i' \boldsymbol{w} - \sum_{i=1}^{n} \lambda_i y_i b$$

- Stationarity conditions give us more information:

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{i=1}^{n} \lambda_i y_i = 0 \Rightarrow$$

New constraint,
Eliminates primal variable b

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \boldsymbol{w}^* - \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i = 0 \Rightarrow$$

Eliminates primal variable

$$\boldsymbol{w}^* = \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i$$

- The Lagrangian becomes (with additional constraint, above)

$$\mathcal{L}(\boldsymbol{w}^*, b, \boldsymbol{\lambda}) = \sum_{i=1}^{n} \lambda_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i' \boldsymbol{x}_j$$

Dual program for hard-margin SVM

- Having minimised the Lagrangian with respect to primal variables, now maximising w.r.t dual variables yields the dual program

$$\underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i' \boldsymbol{x}_j$$

$$\text{s.t. } \lambda_i \geq 0 \text{ and } \textstyle\sum_{i=1}^{n} \lambda_i y_i = 0$$

- Strong duality: Solving dual, solves the primal!!
- Like primal: A so-called *quadratic program* - off-the-shelf software can solve – more later
- Unlike primal:
  * Complexity of solution is O(n^3) instead of O(d^3) – more later
  * Program depends on dot products of data only – more later on kernels!

Making predictions with dual solution

Recovering primal variables

每个点都有个lambda，看哪个大于0

=0的 discard

>0的都是 support vector.

- Recall from stationarity: $\boldsymbol{w}^* = \sum_{i=1}^{n} \lambda_i y_i \boldsymbol{x}_i$
- Complementary slackness: b^* can be recovered from dual solution, noting for any example j with $\lambda_j^* > 0$, we have $y_j\left(b^* + \sum_{i=1}^{n} \lambda_i^* y_i \boldsymbol{x}_i' \boldsymbol{x}_j\right) = 1$ (these are the support vectors)

<u>Testing</u>: classify new instance $\boldsymbol{x}$ based on sign of

$$s = b^* + \sum_{i=1}^{n} \lambda_i^* y_i \boldsymbol{x}_i' \boldsymbol{x}$$

一旦找到λ后就能把所有$\lambda_i=0$的点丢掉

Soft-margin SVM's dual

$\lambda_i y_i = 0$

- Training: find $\boldsymbol{\lambda}$ that solves

$$\underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i' \boldsymbol{x}_j$$

box constraints

s.t. $C \geq \lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$

- Testing: same pattern as in as in hard-margin case

Finally… Training the SVM

- The SVM dual problems are quadratic programs, solved in $O(n^3)$, or $O(d^3)$ for the primal.
- This can be inefficient; specialised solutions exist
  - chunking: original SVM training algorithm exploits fact that many λ_i's will be zero (sparsity)
  - sequential minimal optimisation (SMO), an extreme case of chunking. An iterative procedure that analytically optimises randomly chosen pairs (λ_i, λ_j) per iteration

Mini summary

- Dual vs primal formulation of SVM
- Method of Lagrange Multipliers
- Approaches to make predictions and train

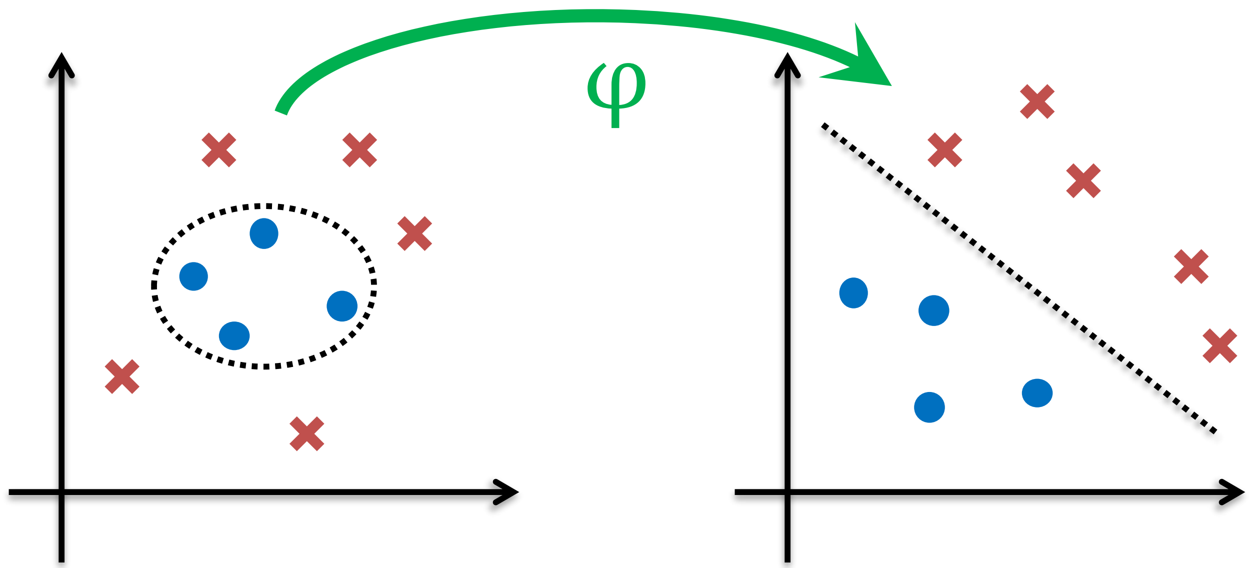
Next: Kernelising the SVM

Kernelising the SVM

Feature transformation by basis expansion; sped up by direct evaluation of kernels – the ‘kernel trick’

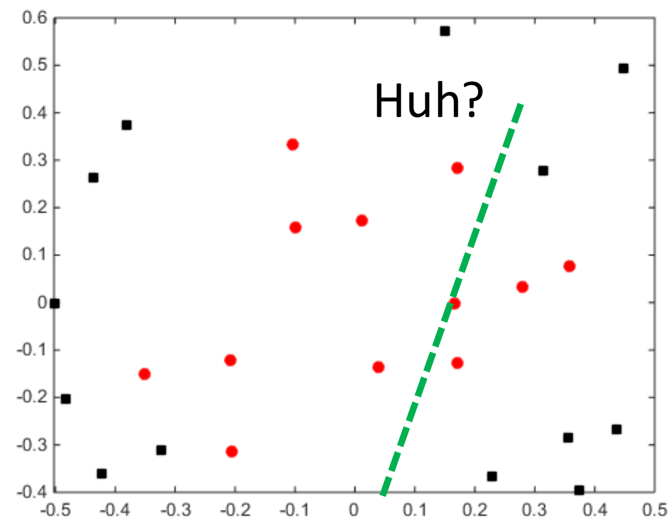
Handling non-linear data with the SVM

- Method 1: Soft-margin SVM
- Method 2: Feature space transformation
  - Map data into a new feature space
  - Run hard-margin or soft-margin SVM in new space
  - Decision boundary is non-linear in original space

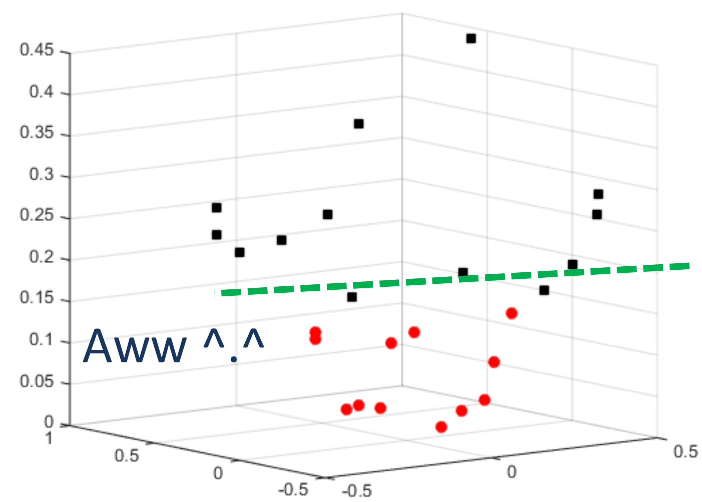


Feature transformation (Basis expansion)

- Consider a binary classification problem
- Each example has features
$$\boldsymbol{x} = [x_1, x_2]$$
- Not linearly separable



- Now 'add' a feature $x_3 = x_1^2 + x_2^2$
- Each point is now
$$\varphi(\boldsymbol{x}) = [x_1, x_2, x_1^2 + x_2^2]$$
- Linearly separable!



Naïve workflow

- Choose/design a linear model
- Choose/design a high-dimensional transformation $\varphi(\boldsymbol{x})$
  - Hoping that after adding a lot of various features some of them will make the data linearly separable
- For each training example, and for each new instance compute $\varphi(\boldsymbol{x})$
- Train classifier/Do predictions

- Problem: impractical/impossible to compute $\varphi(\boldsymbol{x})$ for high/infinite-dimensional $\varphi(\boldsymbol{x})$

Hard-margin SVM's dual formulation

- Training: finding $\boldsymbol{\lambda}$ that solve

dot-product

$$\underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \boldsymbol{x}_i' \boldsymbol{x}_j$$

$$\text{s.t. } \lambda_i \geq 0 \text{ and } \sum_{i=1}^{n} \lambda_i y_i = 0$$

- Making predictions: classify new instance $\boldsymbol{x}$ as sign of

dot-product

$$s = b^* + \sum_{i=1}^{n} \lambda_i^* y_i \boldsymbol{x}_i' \boldsymbol{x}$$

Note: b^* found by solving for it in $y_j\left(b^* + \sum_{i=1}^{n} \lambda_i^* y_i \boldsymbol{x}_i' \boldsymbol{x}_j\right) = 1$ for any support vector j

Hard-margin SVM in *feature space*

- Training: finding $\boldsymbol{\lambda}$ that solve

$$\underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j \varphi(\boldsymbol{x}_i)' \varphi(\boldsymbol{x}_j)$$

$$\text{s.t. } \lambda_i \geq 0 \text{ and } \textstyle\sum_{i=1}^{n} \lambda_i y_i = 0$$

- Making predictions: classify new instance $\boldsymbol{x}$ as sign of

$$s = b^* + \sum_{i=1}^{n} \lambda_i^* y_i \varphi(\boldsymbol{x}_i)' \varphi(\boldsymbol{x})$$

Note: b^* found by solving for it in $y_j\left(b^* + \sum_{i=1}^{n} \lambda_i^* y_i \varphi(\boldsymbol{x}_i)' \varphi(\boldsymbol{x}_j)\right) = 1$ for support vector j

Observation: Kernel representation

- Both parameter estimation and computing predictions depend on data only in a form of a dot product
  - In original space $\boldsymbol{u}'\boldsymbol{v} = \sum_{i=1}^{m} u_i v_i$
  - In transformed space $\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = \sum_{i=1}^{l} \varphi(\boldsymbol{u})_i \varphi(\boldsymbol{v})_i$

- Kernel is a function that can be expressed as a dot product in some feature space $\underbrace{K(\boldsymbol{u}, \boldsymbol{v}) = \varphi(\boldsymbol{u})'\varphi(\boldsymbol{v})}_{\text{Kernel}}$

Kernel as shortcut: Example

- For *some* $\varphi(\boldsymbol{x})$'s, kernel is faster to compute directly than first mapping to feature space then taking dot product.

- E.g., consider two 1-D vectors $\boldsymbol{u} = [u_1]$ and $\boldsymbol{v} = [v_1]$ and transformation $\varphi(\boldsymbol{x}) = [x_1^2, \sqrt{2c}x_1, c]$, some c
  - So $\varphi(\boldsymbol{u}) = \underbrace{\left[u_1^2, \sqrt{2c}u_1, c\right]'}^{\text{2 operations}}$ and $\varphi(\boldsymbol{v}) = \overbrace{\left[v_1^2, \sqrt{2c}v_1, c\right]'}^{\text{+2 operations}}$
  - Then $\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = \left(u_1^2 v_1^2 + 2cu_1v_1 + c^2\right)$ `+5 operations = 9 ops.`

- This can be alternatively computed directly as
$$\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = (u_1 v_1 + c)^2 \quad \text{3 operations}$$
  - Here $K(\boldsymbol{u}, \boldsymbol{v}) = (u_1 v_1 + c)^2$ is the corresponding kernel

直接算核函数就简单 much less operations

More generally: The "kernel trick"

- Consider two training points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ and their dot product in the transformed space.
- $k_{ij} \equiv \varphi(\boldsymbol{x}_i)'\varphi(\boldsymbol{x}_j)$ kernel matrix can be computed as:
  1. Compute $\varphi(\boldsymbol{x}_i)'$
  2. Compute $\varphi(\boldsymbol{x}_j)$
  3. Compute $k_{ij} = \varphi(\boldsymbol{x}_i)'\varphi(\boldsymbol{x}_j)$ kernel is the output of an inner product, inner product is a scalar
- However, for some transformations φ, there's a "shortcut" function that gives exactly the same answer $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = k_{ij}$
  * Doesn't involve steps 1 – 3 and no computation of $\varphi(\boldsymbol{x}_i)$ and $\varphi(\boldsymbol{x}_j)$
  * Usually k_{ij} computable in $O(m)$, but computing $\varphi(\boldsymbol{x})$ requires $O(l)$, where $\boldsymbol{l \gg m}$ **(impractical)** and even $\boldsymbol{l = \infty}$ **(infeasible)**

$K(x_i, x_j)$ takes two data points in some dimension d, outputs a scalar.

it has to be a function that is associated with some inner product of some feature function φ.

φ function

Kernel hard-margin SVM

feature mapping is implied by kernel

- Training: finding $\boldsymbol{\lambda}$ that solve

$$\underset{\boldsymbol{\lambda}}{\operatorname{argmax}} \sum_{i=1}^{n} \lambda_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \lambda_i \lambda_j y_i y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

high dimensional interpretation

s.t. $\lambda_i \geq 0$ and $\sum_{i=1}^{n} \lambda_i y_i = 0$

- Making predictions: classify new instance $\boldsymbol{x}$ based on the sign of

$$s = b^* + \sum_{i=1}^{n} \lambda_i^* y_i K(\boldsymbol{x}_i, \boldsymbol{x})$$

feature mapping is implied by kernel

- Here b^* can be found by noting that for support vector j we have $y_j \left(b^* + \sum_{i=1}^{n} \lambda_i^* y_i K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right)\right) = 1$

Approaches to non-linearity

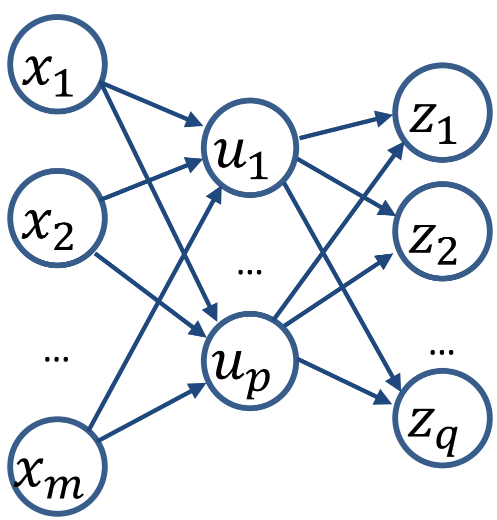
for NN: φ has weights learned from data

for SVM: φ doesnot have weight.

NNets

在NN中:我学 φ function

- Elements of $\boldsymbol{u} = \varphi(\boldsymbol{x})$ are transformed input $\boldsymbol{x}$
- This φ has weights learned from data



SVMs

在这里不用计算 φ

- Choice of kernel K determines features φ
- Don't learn φ weights
- But, don't even need to compute φ so can support v high dim. φ
- Also support arbitrary data types

Mini summary

- Kernelisation
  - Basis expansion on dual formulation of SVMs
  - “Kernel trick”; Fast computation of feature space dot product

Next: Kernel methods as modular machine learning

Modular Learning

Kernelisation beyond SVMs;
Separating the "learning module"
from feature space transformation

Modular learning

- All information about feature mapping is concentrated within the kernel
- In order to use a different feature mapping, simply change the kernel function
- Algorithm design decouples into choosing a “learning method” (e.g., SVM vs logistic regression) and choosing feature space mapping, i.e., kernel
- But how to know if an algorithm is a kernel method?

Representer theorem

Theorem: For any training set $\{\boldsymbol{x}_n, y_n, \ldots, \boldsymbol{x}_n, y_n\}$, any empirical risk function $\hat{R}$, monotonic increasing function g, then any solution

monotonic function square or other

$$f^* \in \arg\min_f \left(\hat{R}(\boldsymbol{x}_1, y_1, f(\boldsymbol{x}_1), \ldots, \boldsymbol{x}_n, y_n, f(\boldsymbol{x}_n)) + g(\|f\|) \right)$$

has representation for some coefficients α_i's

$$f^*(\boldsymbol{x}) = \sum_{i=1}^{n} \underbrace{\alpha_i}_{\text{weights}} K(\boldsymbol{x}, \boldsymbol{x}_i)$$

in SVM $\alpha = \lambda_i y_i$

* Tells us when a (decision-theoretic) learner is kernelizable
* The dual tells us the form this linear kernel representation takes
* SVM not the only case:
  * Ridge regression, Logistic regression
  * Principal component analysis (PCA)
  * Canonical correlation analysis (CCA)
  * Linear discriminant analysis (LDA), and many more…

Mini summary

- Kernel methods are modular
  - Choose learning algorithm
  - Choose kernel
- Representer thm: recognises kernelisable learners

Next: Constructing and recognising kernels

Constructing Kernels

An overview of popular kernels, kernel properties for building and recognising new kernels

Polynomial kernel

- Function $K(\boldsymbol{u}, \boldsymbol{v}) \overset{\varphi(u)\varphi(v)}{=} (\boldsymbol{u}'\boldsymbol{v} + c)^d$ is called *polynomial kernel*
  - Here $\boldsymbol{u}$ and $\boldsymbol{v}$ are vectors with m components
  - $d \geq 0$ is an integer and $c \geq 0$ is a constant

- Without loss of generality, assume $c = 0$
  - If it's not, add $\sqrt{c}$ as a dummy feature to $\boldsymbol{u}$ and $\boldsymbol{v}$

- $(\boldsymbol{u}'\boldsymbol{v})^d = \underbrace{(u_1 v_1 + \dots + u_m v_m) \dots (u_1 v_1 + \dots + u_m v_m)}_{d \text{ times}}$

$= \sum_{i=1}^{l} (u_1 v_1)^{a_{i1}} \dots (u_m v_m)^{a_{im}}$

Here $0 \leq a_{ij} \leq d$ and l are integers

$= \sum_{i=1}^{l} \left(u_1^{a_{i1}} \dots u_m^{a_{im}}\right)' \left(v_1^{a_{i1}} \dots v_m^{a_{im}}\right)$

$= \sum_{i=1}^{l} \varphi_i(\boldsymbol{u}) \varphi_i(\boldsymbol{v})$

E.g., for $d = 2, m = 2$ 2维

$(\boldsymbol{u}'\boldsymbol{v})^2 = (u_1 v_1 + u_2 v_2)(u_1 v_1 + u_2 v_2)$
$= (u_1 v_1)^2 + 2(u_1 v_1)(u_2 v_2) + (u_2 v_2)^2$
$= u_1^2 v_1^2 + 2(u_1 u_2)(v_1 v_2) + u_2^2 v_2^2$
$= \varphi(\boldsymbol{u})\ \varphi(\boldsymbol{v})$

$\varphi(\boldsymbol{u}) = [u_1^2, \sqrt{2} u_1 u_2, u_2^2]$

$\varphi(v) = [v_1^2, \sqrt{2} v_1 v_2, v_2^2]^T$

- Feature map $\varphi: \mathbb{R}^m \to \mathbb{R}^l$, where $\varphi_i(\boldsymbol{x}) = x_1^{a_{i1}} \dots x_m^{a_{im}}$

constant is missing

feature i is some entry of the vector to

$g(u,v) = K(u,v) \times K'(u,v)$

Some power and some entry of the vector to some different power!

$K(u,v) = \varphi(u)\varphi(v)$

$\varphi(u) = (u_1^2 u_2^0, \sqrt{2}u_1^1 u_2^1, u_2^2 u_1^0)$

$K'(u,v) = \varphi'(u)\varphi'(v)$

$$g(u,v) = \varphi(u)\varphi(v)\varphi'(u)\varphi'(v)$$

$$= \varphi'(u)\varphi'(u)\varphi(v)\varphi'(v)$$

$$= \psi(u)\psi(v)$$

where $\psi(u) = \varphi'(u)\varphi'(u)$ and $\psi(v) = \varphi(v)\varphi'(v)$

$$\begin{tikzcd} \boxed{x_i \quad x_j} \arrow[d] \arrow[rr, bend left, "?\ \text{direct}"] & & \\ \phi(x_i) \quad \phi(x_j) \arrow[rr] & & \phi(x_i)^T\phi(x_j) = K_{ij} \end{tikzcd}$$

example $x = [f_1 \ f_2]$
$x' = [g_1 \ g_2]$

$$(x^T x' + 1)^2 = \left([f_1 \ f_2]\begin{bmatrix} g_1 \\ g_2 \end{bmatrix} + 1\right)^2 = (f_1g_1 + f_2g_2 + 1)^2 = f_1^2g_1^2 + f_2^2g_2^2 + 1 + 2f_1g_1f_2g_2 + 2f_1g_1 + 2f_2g_2$$

$$= [f_1^2 \ f_2^2 \ 1 \ \sqrt{2}f_1f_2 \ \sqrt{2}f_1 \ \sqrt{2}f_2]\begin{bmatrix} g_1^2 \\ g_2^2 \\ 1 \\ \sqrt{2}g_1g_2 \\ \sqrt{2}g_1 \\ \sqrt{2}g_2 \end{bmatrix}$$

$$= \phi(x)^T\phi(x')$$

where $\phi(x) = \phi\left(\begin{bmatrix} a \\ b \end{bmatrix}\right) = \begin{bmatrix} a^2 \\ b^2 \\ 1 \\ \sqrt{2}ab \\ \sqrt{2}a \\ \sqrt{2}b \end{bmatrix}$

$(x^Tx'+1)^2$ computes the dot-product in a "transformed space".

$$x \in R^2 \rightarrow \phi(x) \in R^6$$

prove Valid kernel: $\exists \phi: R^d \to R^D$ s.t $k(x, x') = \phi(x)^T \phi(x')$

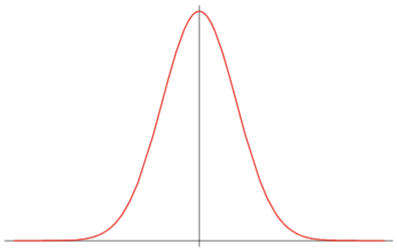
Identifying new kernels

- Method 1: Let $K_1(\boldsymbol{u},\boldsymbol{v})$, $K_2(\boldsymbol{u},\boldsymbol{v})$ be kernels, $c > 0$ be a constant, and $f(\boldsymbol{x})$ be a real-valued function. Then each of the following is also a kernel:
  - $K(\boldsymbol{u},\boldsymbol{v}) = K_1(\boldsymbol{u},\boldsymbol{v}) + K_2(\boldsymbol{u},\boldsymbol{v})$ Summation
  - $K(\boldsymbol{u},\boldsymbol{v}) = cK_1(\boldsymbol{u},\boldsymbol{v})$ multiply with a constant
  - $K(\boldsymbol{u},\boldsymbol{v}) = f(\boldsymbol{u})K_1(\boldsymbol{u},\boldsymbol{v})f(\boldsymbol{v})$ multiply a function ~~with~~ of u at lhs then multiply a function of v at Rhs
  - *See Bishop for more identities*

- Method 2: Using Mercer's theorem (coming up!)

Radial basis function kernel

- Function $K(\boldsymbol{u},\boldsymbol{v}) = \exp(-\gamma\|\boldsymbol{u}-\boldsymbol{v}\|^2)$ is the ***radial basis function kernel*** (aka Gaussian kernel)
  - Here $\gamma > 0$ is the spread parameter



- $\exp(-\gamma\|\boldsymbol{u}-\boldsymbol{v}\|^2) = \exp\left(-\gamma(\boldsymbol{u}-\boldsymbol{v})'(\boldsymbol{u}-\boldsymbol{v})\right)$

$$= \exp\left(-\gamma(\boldsymbol{u}'\boldsymbol{u} - 2\boldsymbol{u}'\boldsymbol{v} + \boldsymbol{v}'\boldsymbol{v})\right)$$

$$= \exp(-\gamma\boldsymbol{u}'\boldsymbol{u})\exp(2\gamma\boldsymbol{u}'\boldsymbol{v})\exp(-\gamma\boldsymbol{v}'\boldsymbol{v})$$

$$= f(\boldsymbol{u})\exp(2\gamma\boldsymbol{u}'\boldsymbol{v})f(\boldsymbol{v})$$

Taylor series expansion:

$$e^z = \sum_{d=0}^{\infty}\frac{z^d}{d!} = 1 + z + \frac{z^2}{2!} + \cdots$$

$$= f(\boldsymbol{u})\left(1 + 2\gamma\boldsymbol{u}'\boldsymbol{v} + 2\gamma^2(\boldsymbol{u}'\boldsymbol{v})^2 + \cdots\right)f(\boldsymbol{v})$$

polynomial kernel: $(u'v)^0 + (u'v)^1 \quad (u'v)^2$

  - Each $(\boldsymbol{u}'\boldsymbol{v})^d$ is a polynomial kernel. Using kernel identities, the middle term is a kernel, and hence the whole expression is a kernel

Mercer's Theorem

- Question: given $\varphi(\boldsymbol{u})$, is there a good kernel to use?
- Inverse question: given some function $K(\boldsymbol{u}, \boldsymbol{v})$, is this a valid kernel? In other words, is there a mapping $\varphi(\boldsymbol{u})$ implied by the kernel?

- Mercer's theorem:
  * Consider a finite sequence of objects $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$
  * Construct $n \times n$ matrix of pairwise values
  $$M_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$
  * K is a valid kernel if matrix M is positive-semidefinite, for all possible sequences $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$

Handling arbitrary data structures

- Kernels are powerful approach to deal with many data types
- Could define similarity function on variable length strings

K(“science is organized knowledge”, “wisdom is organized life”)

- However, not every function on two objects is a valid kernel
- Remember that we need that function $K(\boldsymbol{u}, \boldsymbol{v})$ to imply a dot product in some feature space

A large variety of kernels

Definition 9.1 Polynomial kernel 286
Computation 9.6 All-subsets kernel 289
Computation 9.8 Gaussian kernel 290
Computation 9.12 ANOVA kernel 293
Computation 9.18 Alternative recursion for ANOVA kernel 296
Computation 9.24 General graph kernels 301
Definition 9.33 Exponential difusion kernel 307
Definition 9.34 von Neumann difusion kernel 307
Computation 9.35 Evaluating difusion kernels 308
Computation 9.46 Evaluating randomised kernels 315
Definition 9.37 Intersection kernel 309
Definition 9.38 Union-complement kernel 310
Remark 9.40 Agreement kernel 310
Section 9.6 Kernels on real numbers 311
Remark 9.42 Spline kernels 313
Definition 9.43 Derived subsets kernel 313
Definition 10.5 Vector space kernel 325
Computation 10.8 Latent semantic kernels 332
Definition 11.7 The p-spectrum kernel 342
Computation 11.10 The p-spectrum recursion 343
Remark 11.13 Blended spectrum kernel 344
Computation 11.17 All-subsequences kernel 347
Computation 11.24 Fixed length subsequences kernel 352
Computation 11.33 Naive recursion for gap-weighted subsequences kernel 358
Computation 11.36 Gap-weighted subsequences kernel 360
Computation 11.45 Trie-based string kernels 367
Algorithm 9.14 ANOVA kernel 294
Algorithm 9.25 Simple graph kernels 302
Algorithm 11.20 All–non-contiguous subsequences kernel 350
Algorithm 11.25 Fixed length subsequences kernel 352
Algorithm 11.38 Gap-weighted subsequences kernel 361
Algorithm 11.40 Character weighting string kernel 364
Algorithm 11.41 Soft matching string kernel 365
Algorithm 11.42 Gap number weighting string kernel 366
Algorithm 11.46 Trie-based p-spectrum kernel 368
Algorithm 11.51 Trie-based mismatch kernel 371
Algorithm 11.54 Trie-based restricted gap-weighted kernel 374
Algorithm 11.62 Co-rooted subtree kernel 380
Algorithm 11.65 All-subtree kernel 383
Algorithm 12.8 Fixed length HMM kernel 401
Algorithm 12.14 Pair HMM kernel 407
Algorithm 12.17 Hidden tree model kernel 411
Algorithm 12.34 Fixed length Markov model Fisher kernel 427

www.kernel-methods.net

Mini Summary

- Constructing kernels
  - An overview of popular kernels and their properties
  - Mercer's theorem
  - Extending machine learning beyond conventional data structure

Next lecture: Perceptron