Lecture 14 Kernel Methods

COMP90051 Statistical Machine Learning

Semester 1, 2021
Lecturer: Trevor Cohn

This lecture

- Dual formulation of 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:

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

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

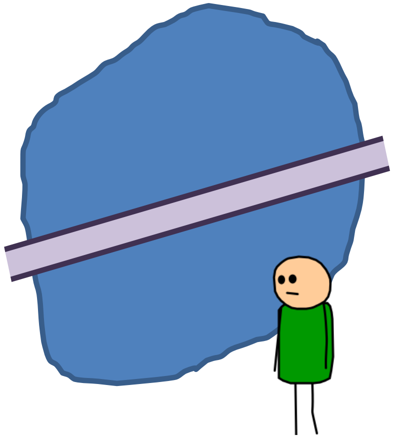
$$\xi_i \geq 0 \text{ for } i = 1, \dots, n$$

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

*Changed $\|\boldsymbol{w}\|$ to $0.5\|\boldsymbol{w}\|^2$ - monotonic increasing transform. Modified objective yields same solution.

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. $1 - y_i(\boldsymbol{w}'\boldsymbol{x}_i + b) \leq 0$ for $i = 1, \dots, n$

- 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

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

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

May be easier to solve, advantageous

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

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:
  * $g_i(\boldsymbol{x}^*) \leq 0, i = 1, \dots, n$
  * $h_j(\boldsymbol{x}^*) = 0, j = 1, \dots, m$

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

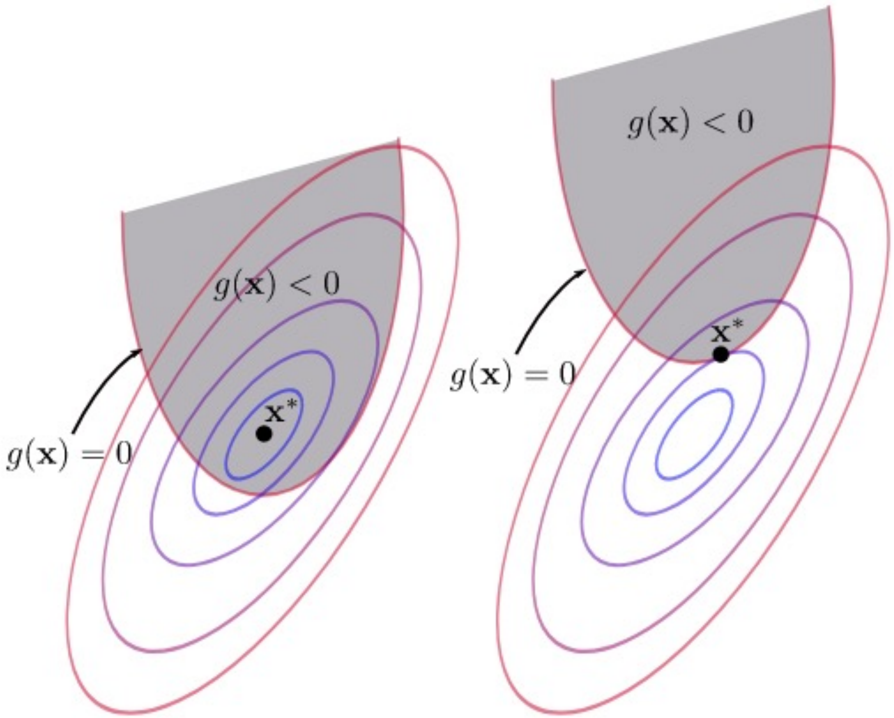
- Dual feasibility: $\lambda_i^* \geq 0$ for $i = 1, \dots, n$
- Complementary slackness: $\lambda_i^* g_i(\boldsymbol{x}^*) = 0, i = 1, \dots, n$

Don't penalise if constraint satisfied

- Stationarity: $\nabla_x \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) = \boldsymbol{0}$

KKT conditions example

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



KKT conditions for hard-margin SVM

The Lagrangian

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

KKT conditions:

* Feasibility: $y_i\left((\boldsymbol{w}^*)'\boldsymbol{x}_i + b^*\right) - 1 \geq 0$ for $i = 1, \dots, n$
* Feasibility: $\lambda_i^* \geq 0$ for $i = 1, \dots, n$
* Complementary slackness: $\lambda_i^*\left(y_i\left((\boldsymbol{w}^*)'\boldsymbol{x}_i + b^*\right) - 1\right) = 0$
* Stationarity: $\nabla_{\boldsymbol{w},b}\mathcal{L}(\boldsymbol{w}^*, b^*, \boldsymbol{\lambda}^*) = \boldsymbol{0}$

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

- Lagrangian:

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

- Stationarity conditions give us more information:

$$\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^{n} \lambda_i y_i = 0 \quad \Rightarrow \quad \text{New constraint}$$

$$\frac{\partial \mathcal{L}}{\partial w_j} = w_j^* - \sum_{i=1}^{n} \lambda_i y_i (\boldsymbol{x}_i)_j = 0 \quad \Rightarrow \quad \text{Eliminates primal variables}$$

- The Lagrangian becomes (with additional constraint, above)

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

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

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

Testing: classify new instance $\boldsymbol{x}$ based on sign of

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

Soft-margin SVM's dual

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

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

box constraints

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

- Making predictions: 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 inefficient; Several specialised solutions proposed:
  - chunking: original SVM training algorithm exploits fact that many λs will be zero (sparsity)
  - sequential minimal optimisation (SMO), an extreme case of chunking. An iterative procedure that analytically optimises randomly chosen pairs of λs per iteration

Mini summary

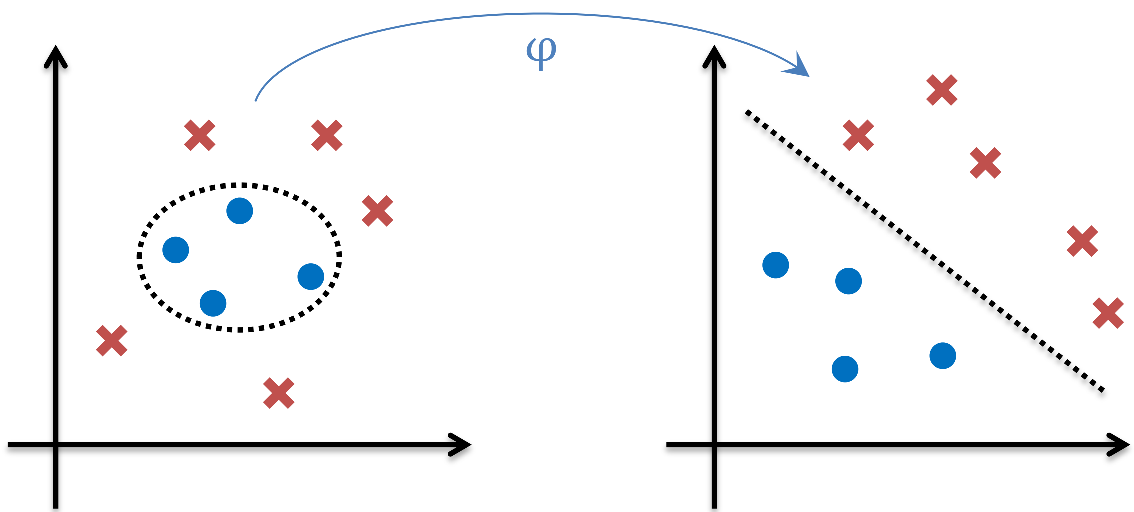
- Dual vs primal formulation of SVM
- Method of Lagrange Multipliers
- Means of prediction, training algorithms

Kernelising the SVM

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

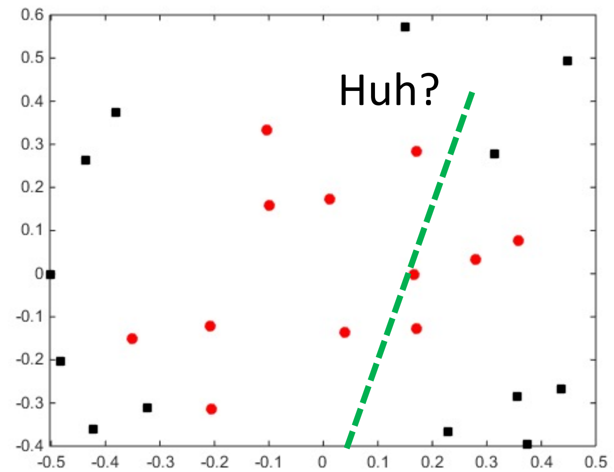
Handling non-linear data with the SVM

- Method 1: Soft-margin SVM (Lect 13)
- Method 2: Feature space transformation (Lect 3)
  - 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

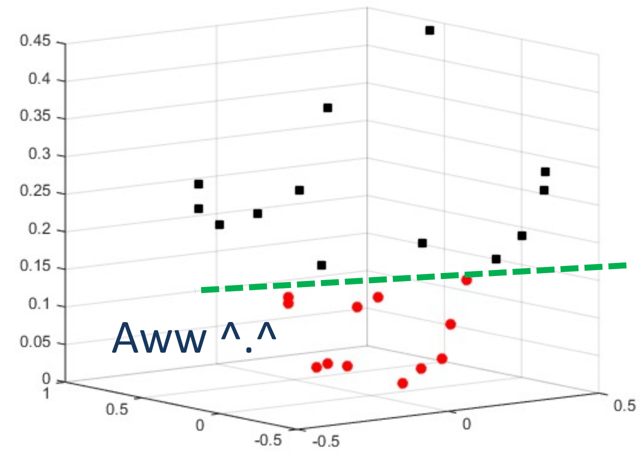


Feature transformation (Basis expansion)

- Consider a binary classification problem
- Each example has features $[x_1, x_2]$
- Not linearly separable



- Now 'add' a feature $x_3 = x_1^2 + x_2^2$
- Each point is now $[x_1, x_2, x_1^2 + x_2^2]$
- 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

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

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

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

dot-product

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

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

Hard-margin SVM in *feature space*

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

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

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

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

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

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

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.

- For example, consider two 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 $\varphi(\boldsymbol{u}) = \underbrace{\left[u_1^2, \sqrt{2c}u_1, c\right]'}$ and $\varphi(\boldsymbol{v}) = \left[v_1^2, \sqrt{2c}v_1, c\right]'$

    2 operations   +2 operations
  - Then $\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = \left(u_1^2 v_1^2 + 2cu_1v_1 + c^2\right)$ +4 operations = 8 ops.

- This can be alternatively computed directly as

$$\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = (u_1v_1 + c)^2$$

3 operations

  - Here $K(\boldsymbol{u}, \boldsymbol{v}) = (u_1v_1 + c)^2$ is the corresponding kernel

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 $k_{ij} = \varphi(\boldsymbol{x}_i)'\varphi(\boldsymbol{x}_j)$
- 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 $l \gg m$ (impractical) and even $l = \infty$ (infeasible)

Kernel hard-margin SVM

feature mapping is implied by kernel

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

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

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

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

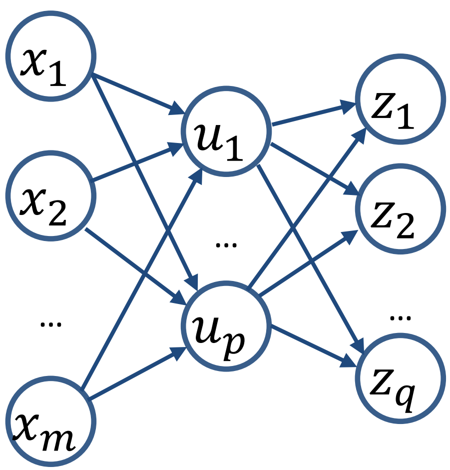
feature mapping is implied by kernel

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

Approaches to non-linearity

ANNs

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



SVMs

- 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

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

Kernelised perceptron (1/3)

When classified correctly, weights are unchanged

When misclassified: $\boldsymbol{w}^{(k+1)} = -\eta(\pm\boldsymbol{x})$
($\eta > 0$ is called *learning rate)*

<u>If $y = 1$, but $s < 0$</u>
$w_i \leftarrow w_i + \eta x_i$
$w_0 \leftarrow w_0 + \eta$

<u>If $y = -1$, but $s \geq 0$</u>
$w_i \leftarrow w_i - \eta x_i$
$w_0 \leftarrow w_0 - \eta$

Suppose weights are initially set to 0

First update: $\boldsymbol{w} = \eta y_{i_1} \boldsymbol{x}_{i_1}$
Second update: $\boldsymbol{w} = \eta y_{i_1} \boldsymbol{x}_{i_1} + \eta y_{i_2} \boldsymbol{x}_{i_2}$
Third update $\boldsymbol{w} = \eta y_{i_1} \boldsymbol{x}_{i_1} + \eta y_{i_2} \boldsymbol{x}_{i_2} + \eta y_{i_3} \boldsymbol{x}_{i_3}$
etc.

Kernelised perceptron (2/3)

- Weights always take the form $\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i$, where $\boldsymbol{\alpha}$ some coefficients
- Perceptron weights always linear comb. of data!
- Recall that prediction for a new point $\boldsymbol{x}$ is based on sign of $w_0 + \boldsymbol{w}'\boldsymbol{x}$
- Substituting $\boldsymbol{w}$ we get $w_0 + \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i'\boldsymbol{x}$
- The dot product $\boldsymbol{x}_i'\boldsymbol{x}$ can be replaced with a kernel

Kernelised perceptron (3/3)

Choose initial guess $\boldsymbol{w}^{(0)}$, $k = 0$

Set $\boldsymbol{\alpha} = \mathbf{0}$

For t from 1 to T (epochs)

  For each training example $\{\boldsymbol{x}_i, y_i\}$

    Predict based on $w_0 + \sum_{j=1}^{n} \alpha_j y_j \boldsymbol{x}_i' \boldsymbol{x}_j$

    If misclassified, <u>update</u> each $\alpha_j \leftarrow \alpha_j + \eta y_j$

Kernelised perceptron (3/3)

Choose initial guess $\boldsymbol{w}^{(0)}, k = 0$

Set $\boldsymbol{\alpha} = \mathbf{0}$

For t from 1 to T (epochs)

For each training example $\{\boldsymbol{x}_i, y_i\}$

Predict based on $w_0 + \sum_{j=1}^{n} \alpha_j y_j K(\boldsymbol{x}_i, \boldsymbol{x}_j)$

If misclassified, <u>update</u> each $\alpha_j \leftarrow \alpha_j + \eta y_j$

Becomes kernel matrix k_{ij}

Representer theorem

- Theorem: For any training set $\{\boldsymbol{x}_i, y_i\}_{i=1}^n$, any empirical risk function E, monotonic increasing function g, then any solution
- $f^* \in \arg\min_f E(\boldsymbol{x}_1, y_1, f(\boldsymbol{x}_1), \dots, \boldsymbol{x}_n, y_n, f(\boldsymbol{x}_n)) + g(\|f\|)$
  - has representation for some coefficients
  - $f^*(\boldsymbol{x}) = \sum_{i=1}^n \alpha_i\, k(\boldsymbol{x}, \boldsymbol{x}_i)$
- Tells us when a (decision-theoretic) learner is kernelizable
- The dual tells us the form this linear kernel representation takes
- SVM & Perceptron not the only cases:
  - Ridge regression
  - Logistic regression
  - Principal component analysis (PCA)
  - Canonical correlation analysis (CCA)
  - Linear discriminant analysis (LDA)
  - and many more …

Constructing Kernels

An overview of popular kernels
and kernel properties

Polynomial kernel

- Function $K(\boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{u}'\boldsymbol{v} + c)^d$ 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 the loss of generality, assume $c = 0$
  - If it's not, add $\sqrt{c}$ as a dummy feature to $\boldsymbol{u}$ and $\boldsymbol{v}$
- $(\boldsymbol{u}'\boldsymbol{v})^d = (u_1 v_1 + \cdots + u_m v_m)(u_1 v_1 + \cdots + u_m v_m) \ldots (u_1 v_1 + \cdots + u_m v_m)$

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

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

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

  $= \sum_{i=1}^{l} \varphi(\boldsymbol{u})_i \varphi(\boldsymbol{v})_i$
- Feature map $\varphi: \mathbb{R}^m \rightarrow \mathbb{R}^l$, where $\varphi_i(\boldsymbol{x}) = \left(x_1^{a_{i1}} \ldots x_m^{a_{im}}\right)$

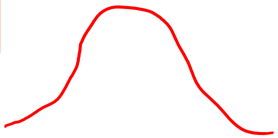
Identifying new kernels

- 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:
  - $K(\boldsymbol{u}, \boldsymbol{v}) = K_1(\boldsymbol{u}, \boldsymbol{v}) + K_2(\boldsymbol{u}, \boldsymbol{v})$
  - $K(\boldsymbol{u}, \boldsymbol{v}) = cK_1(\boldsymbol{u}, \boldsymbol{v})$
  - $K(\boldsymbol{u}, \boldsymbol{v}) = f(\boldsymbol{u})K_1(\boldsymbol{u}, \boldsymbol{v})f(\boldsymbol{v})$
  - *See Bishop for more identities*

Prove these!

- 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

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

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

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

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

$$= f(\boldsymbol{u})\left(\textstyle\sum_{d=0}^{\infty} r_d(\boldsymbol{u}'\boldsymbol{v})^d\right)f(\boldsymbol{v})$$

Power series expansion

- Here, each $(\boldsymbol{u}'\boldsymbol{v})^d$ is a polynomial kernel. Using kernel identities, we conclude that 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 sequences of objects $\boldsymbol{x}_1, \dots, \boldsymbol{x}_n$
  - Construct $n \times n$ matrix of pairwise values $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$
  - $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is a valid kernel if this matrix is positive-semidefinite, and this holds 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

This lecture

- Kernels
  - Nonlinearity by basis expansion
  - Kernel trick to speed up computation
- Modular learning
  - Separating “learning module” from feature transformation
  - Representer theorem
- Constructing kernels
  - An overview of popular kernels and their properties
  - Mercer’s theorem
  - Extending machine learning beyond conventional data structure

Next lecture: Bandits!