Lecture 9. Kernel Methods

COMP90051 Statistical Machine Learning

Lecturer: Jean Honorio

THE UNIVERSITY OF MELBOURNE

Soft-margin SVM recap

- Soft-margin SVM objective:

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

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

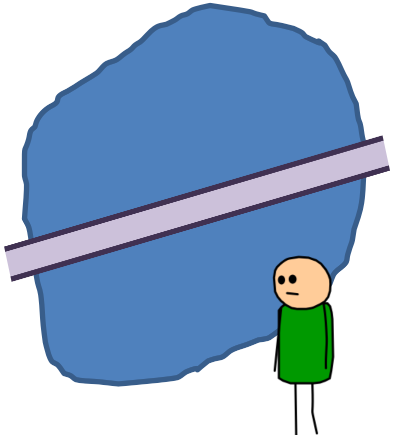
$$\xi_i \geq 0 \text{ for } i = 1, \dots, n$$

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

Constrained optimisation

- Constrained optimisation: canonical form

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



  * 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. $1 - y_i(\boldsymbol{w}'\boldsymbol{x}_i + b) \le 0,$

  $i = 1, \dots, n$

- 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

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

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 $\leq$ primal optimum
  - For convex programs (inc. SVM!) strong duality: optima coincide!

Karush-Kuhn-Tucker Necessary Conditions

- Lagrangian: $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{v}) = f(\boldsymbol{x}) + \sum_{i=1}^{n} \lambda_i g_i(\boldsymbol{x}) + \sum_{j=1}^{m} \nu_j h_j(\boldsymbol{x})$
- 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: $\nabla_{\boldsymbol{x}} \mathcal{L}(\boldsymbol{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{v}^*) = \boldsymbol{0}$

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

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:

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

New constraint,
Eliminates primal variable b

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

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

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

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

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

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

Soft-margin SVM's dual

- 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

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

- 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.

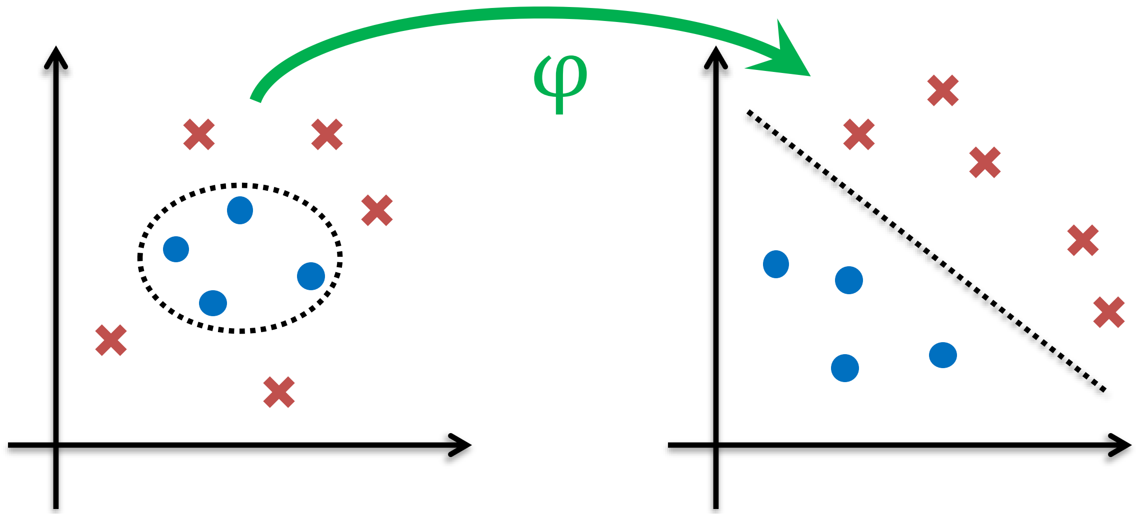
- This can be inefficient; specialised solutions exist
  - chunking: original SVM training algorithm exploits fact that many λ_i's will be zero (sparsity)
  - sequential minimal optimisation (SMO), an extreme case of chunking. An iterative procedure that analytically optimises randomly chosen pairs (λ_i, λ_j) per iteration

Kernelising the SVM

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

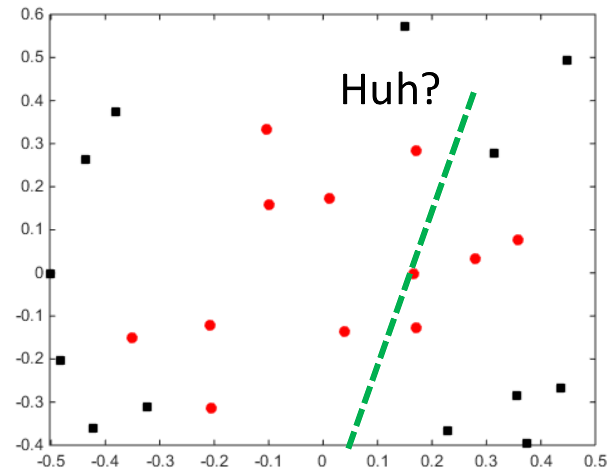
Handling non-linear data with the SVM

- 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

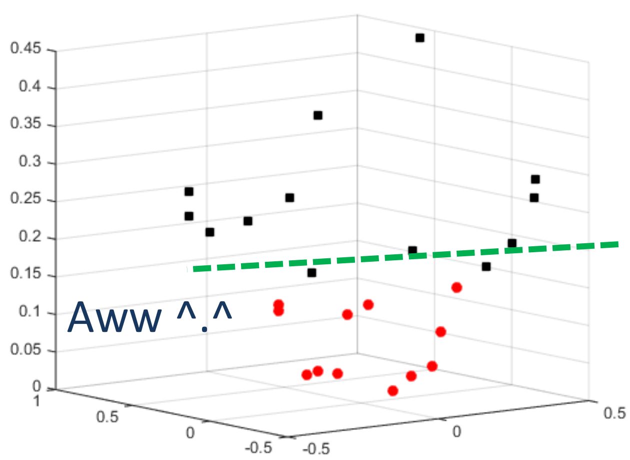


Feature transformation (Basis expansion)

- 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

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

s.t. $\lambda_i \geq 0$ 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

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

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

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

- 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.

- 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 $\varphi(\boldsymbol{u}) = \underbrace{[u_1^2, \sqrt{2c}u_1, c]'}_{\text{2 operations}}$ and $\varphi(\boldsymbol{v}) = \underbrace{[v_1^2, \sqrt{2c}v_1, c]'}_{\text{+2 operations}}$
  - Then $\varphi(\boldsymbol{u})'\varphi(\boldsymbol{v}) = (u_1^2 v_1^2 + 2cu_1 v_1 + c^2)$ +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

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 $\boldsymbol{l \gg m}$ **(impractical)** and even $\boldsymbol{l = \infty}$ **(infeasible)**

Kernel hard-margin SVM

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

feature mapping is implied by kernel

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

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

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

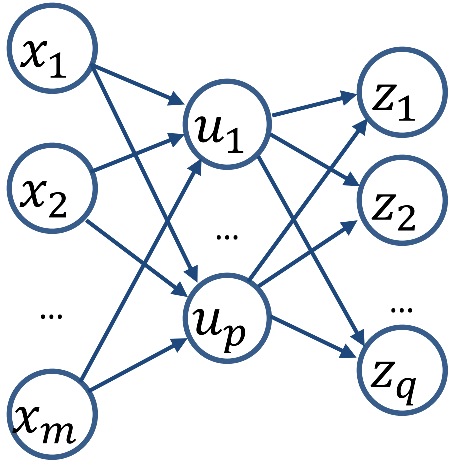
feature mapping is implied by kernel

- 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

NNets

- 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

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 $\{\boldsymbol{x}_n, y_n, \dots, \boldsymbol{x}_n, y_n\}$, any empirical risk function $\hat{R}$, monotonic increasing function g, then any solution

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

has representation for some coefficients α_i's

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

* Tells us when a (decision-theoretic) learner is kernelizable
* The dual tells us the form this linear kernel representation takes
* SVM not the only case:
  - Reformulate the algorithm such that all computations are expressed as inner products, replace inner products with a kernel function
  - If objective function is a combination of empirical loss and a regularization term that depends on the norm of the parameters the optimal solution can be expressed as a linear combination of inner products of the input samples.

Constructing Kernels

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

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 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 $d = 2, m = 2$

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

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

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

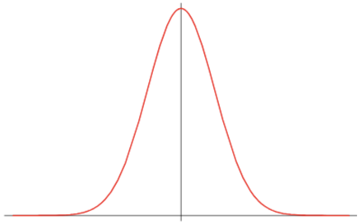
Identifying new kernels

- 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*

- 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



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

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

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

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

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

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

  * 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$
  * 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 $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$

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