Support Vector Machines and Kernels for Language Processing

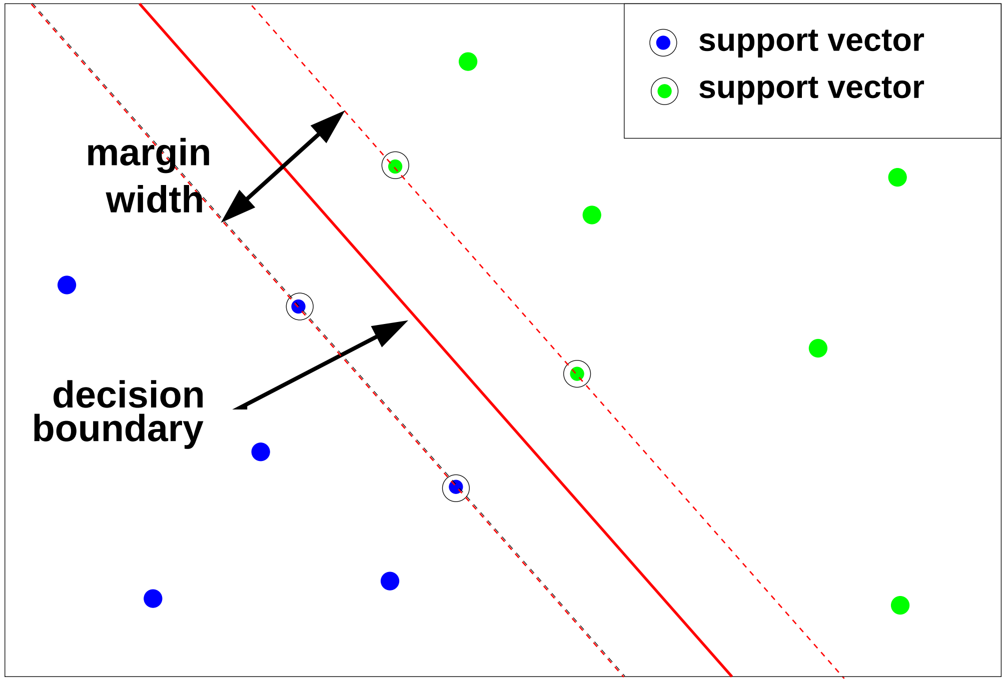
Mark Gales

Lent 2014

Machine Learning for Language Processing: Lecture 7

MPhil in Advanced Computer Science

Support Vector Machines



- SVMs are a maximum margin, binary, classifier:
  - related to minimising generalisation error;
  - unique solution (compare to neural networks);
  - may be kernelised - training/classification a function of dot-product ($\mathbf{x}_i^\top \mathbf{x}_j$).
- Successfully applied to many tasks - how to apply to speech and language?

Training SVMs

- The training criterion can be expressed as

$$\{\hat{\mathbf{w}}, \hat{b}\} = \underset{\mathbf{w},b}{\operatorname{argmax}} \left\{ \min \left\{ ||\boldsymbol{x} - \boldsymbol{x}_i||; \mathbf{w}^\mathsf{T}\boldsymbol{x} + b = 0, i = 1, \ldots, n \right\} \right\}$$

- This can be expressed as the constrained optimisation ($y_i \in \{-1, 1\}$)

$$\{\hat{\mathbf{w}}, \hat{b}\} = \underset{\mathbf{w},b}{\operatorname{argmin}} \left\{ \frac{1}{2}||\mathbf{w}||^2 \right\} \quad \text{subject to } y_i\left(\mathbf{w}^\mathsf{T}\boldsymbol{x}_i + b\right) \geq 1 \quad \forall i$$

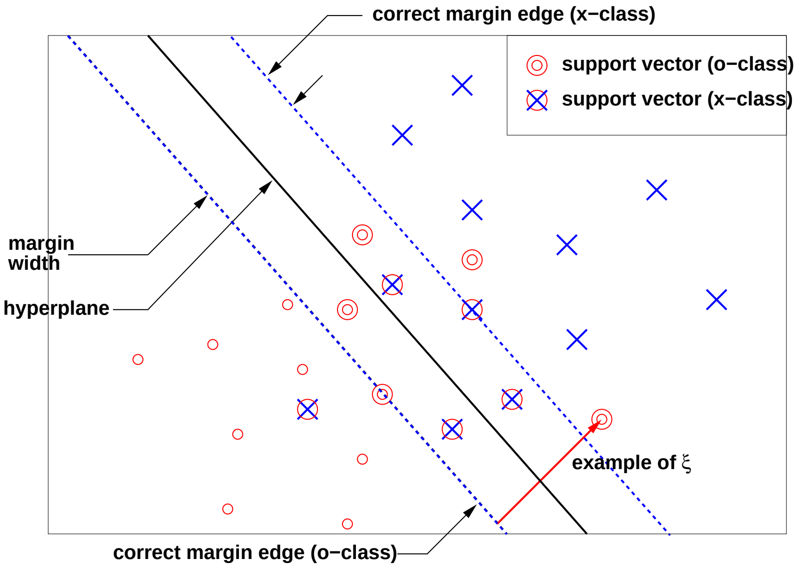
- In practice the dual is optimised

$$\hat{\boldsymbol{\alpha}} = \underset{\boldsymbol{\alpha}}{\operatorname{argmax}} \left\{ \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i^\mathsf{T} \boldsymbol{x}_j \right\}, \quad \hat{\mathbf{w}} = \sum_{i=1}^{n} \hat{\alpha}_i y_i \boldsymbol{x}_i$$

subject to $\alpha_i \geq 0$ and $\sum_{i=1}^{n} \alpha_i y_i = 0$ ($\hat{b}$ is determined given the values of $\hat{\boldsymbol{\alpha}}$)

Non-Separable Data

- Data is not always linearly separable - there's no margin!
  - how to train a system in this (realistic) scenario



- Introduce slack-variables
  - for each training sample $\boldsymbol{x}_i$ introduce ξ_i
  - relaxes constraint: $y_i\left(\mathbf{w}^{\mathsf{T}}\boldsymbol{x}_i + b\right) \geq 1 - \xi_i$
- Modifies the training criterion to be constraints: $y_i\left(\mathbf{w}^{\mathsf{T}}\boldsymbol{x}_i + b\right) \geq 1 - \xi_i, \quad \xi_i \geq 0$

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

- Tunable parameter C - balances margin and upper-bound on training errors
  - again dual form is optimised, but now constraint modified to be: $0 \leq \alpha_i \leq C$

Classification with SVMs

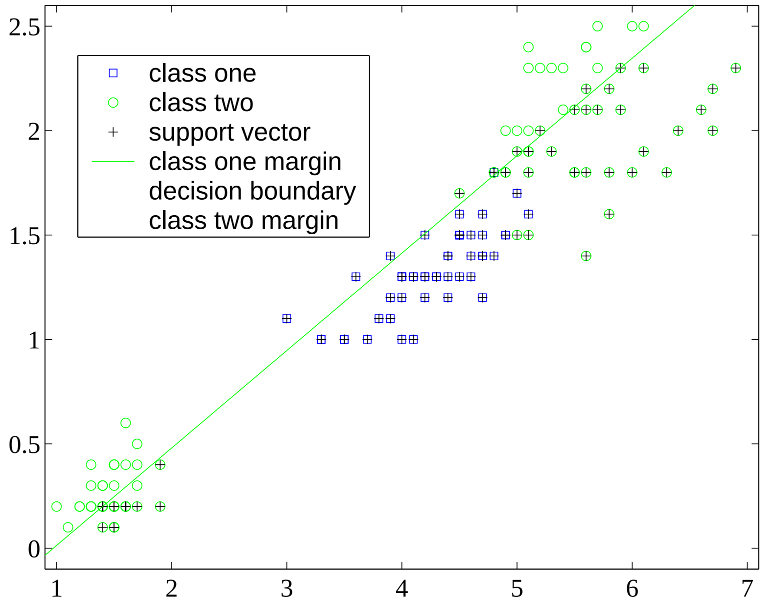
- Given trained parameters $\boldsymbol{\alpha}$ and b classification is based on

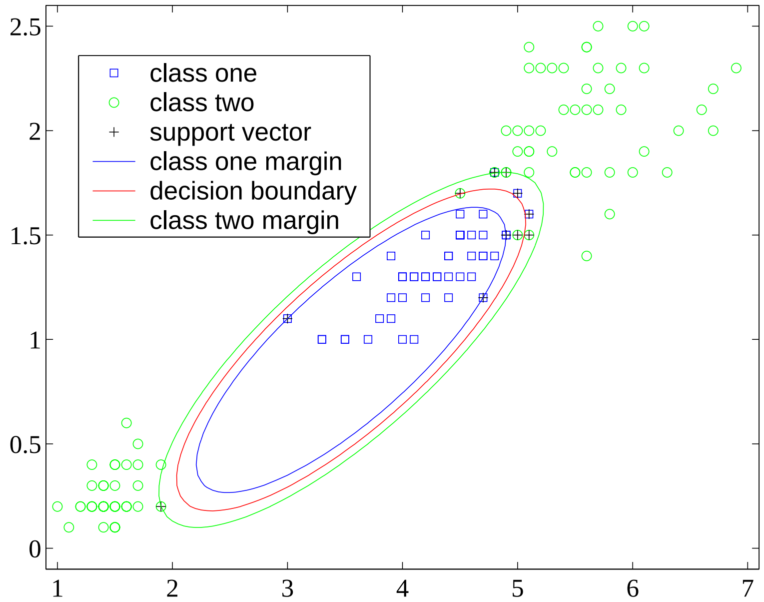
$$g(\boldsymbol{x}) = \mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b = \sum_{i=1}^{n} y_i\alpha_i\boldsymbol{x}_i^{\mathsf{T}}\boldsymbol{x} + b, \quad \hat{\omega} = \begin{cases} \omega_1, & \text{if } g(\boldsymbol{x}) > 0 \\ \omega_2, & \text{otherwise} \end{cases}$$

  - this yields a linear decision boundary - limited
  - classification is based on observations where $\alpha_i > 0$ - the support vectors
- Consider a non-linear transform of the features $\boldsymbol{\phi}(\boldsymbol{x})$ - the feature-space
  - a linear decision boundary in the feature-space is non-linear in original space
- Training and classification can then be implemented in this transformed space
  - classification again based on the support vectors

$$g(\boldsymbol{x}) = \mathbf{w}^{\mathsf{T}}\boldsymbol{\phi}(\boldsymbol{x}) + b = \sum_{i=1}^{n} y_i\alpha_i\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathsf{T}}\boldsymbol{\phi}(\boldsymbol{x}) + b, \quad \hat{\omega} = \begin{cases} \omega_1, & \text{if } g(\boldsymbol{x}) > 0 \\ \omega_2, & \text{otherwise} \end{cases}$$

The "Kernel Trick"





- Consider a simple mapping from a 2-dimensional to 3-dimensional space

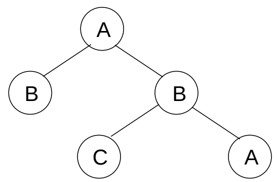
$$\phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}, \quad k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_j)$$

- Efficiently implemented using a Kernel: $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\mathbf{x}_i)^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}_j) = (\boldsymbol{x}_i^\mathsf{T}\boldsymbol{x}_j)^2$

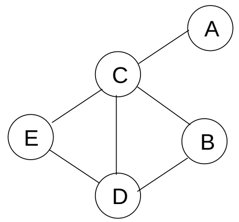
Kernels for Language Processing

- Many standard kernels for fixed length feature vectors
- In language processing applications, data is not always represented by vectors

`... cat sat on the mat ..` word sequences (variable length sequences)



trees (for example parse trees)



graphs showing connections between variables

- Different kernels are used depending on the structures being compared
  - many are based on convolutional kernels
  - an important consideration is the computational cost for particular form

String Kernel

- For sequences input space has variable dimension:
  - use a kernel to map from variable to a fixed length;
  - Fisher kernels are one example for acoustic modelling;
  - String kernels are an example for text.
- Consider the words cat, cart, bar and a character string kernel

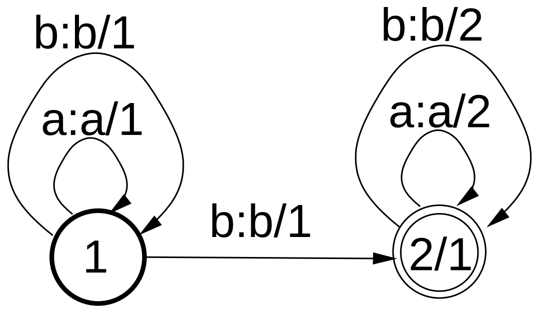
	c-a	c-t	c-r	a-r	r-t	b-a	b-r
$\phi(\texttt{cat})$	1	λ	0	0	0	0	0
$\phi(\texttt{cart})$	1	λ^2	λ	1	1	0	0
$\phi(\texttt{bar})$	0	0	0	1	0	1	λ

$$k(\texttt{cat}, \texttt{cart}) = 1 + \lambda^3, \quad k(\texttt{cat}, \texttt{bar}) = 0, \quad k(\texttt{cart}, \texttt{bar}) = 1$$

- Successfully applied to various text classification tasks:
  - how to make process efficient (and more general)?

Weighted Finite-State Transducers

- A weighted finite-state transducer is a weighted directed graph:
  - transitions labelled with an input symbol, output symbol, weight.
- An example transducer, T, for calculating binary numbers: a=0, b=1



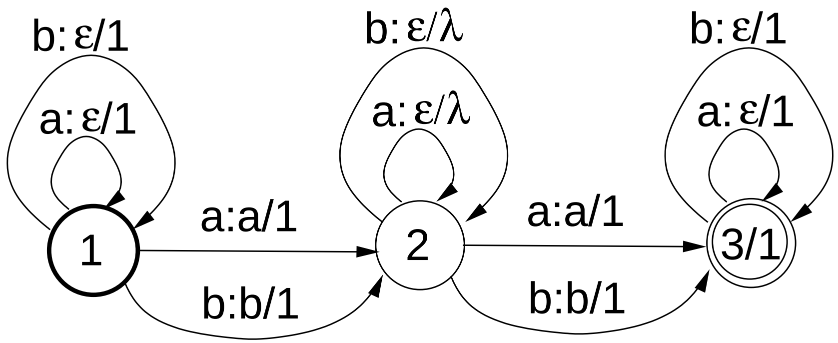
Input	State Seq.	Output	Weight
bab	1 1 2	bab	1
	1 2 2	bab	4

For this sequence output weight: $\texttt{wgt}\left[\texttt{bab} \circ \texttt{T}\right] = 5$

- Standard (highly efficient) algorithms exist for various operations:
  - combining transducer, $\texttt{T}_1 \circ \texttt{T}_2$;
  - inverse, $\texttt{T}^{-1}$, swap the input and output symbols in the transducer.
- May be used for efficient implementation of string kernels.

Rational Kernels

- A transducer, $\mathtt{T}$, for the string kernel (gappy bigram) (vocab $\{\mathtt{a},\mathtt{b}\}$)



The kernel is: $k(\boldsymbol{w}_i, \boldsymbol{w}_j) = \mathtt{wgt}\left[\boldsymbol{w}_i \circ (\mathtt{T} \circ \mathtt{T}^{-1}) \circ \boldsymbol{w}_j\right]$

- This form can also handle uncertainty in decoding ($\boldsymbol{w} = w_1, \ldots, w_N$):
  - lattices can be used rather than the 1-best output.
- This form encompasses various standard feature-spaces and kernels:
  - bag-of-words and N-gram counts, gappy N-grams (string Kernel),
- Successfully applied to a multi-class call classification task.

Tree Kernels

- Tree kernels count the numbers of shared subtrees between trees $\mathcal{T}_1$ and $\mathcal{T}_2$
  - the feature-space, $\phi\left(\mathcal{T}_1\right)$, can be defined as

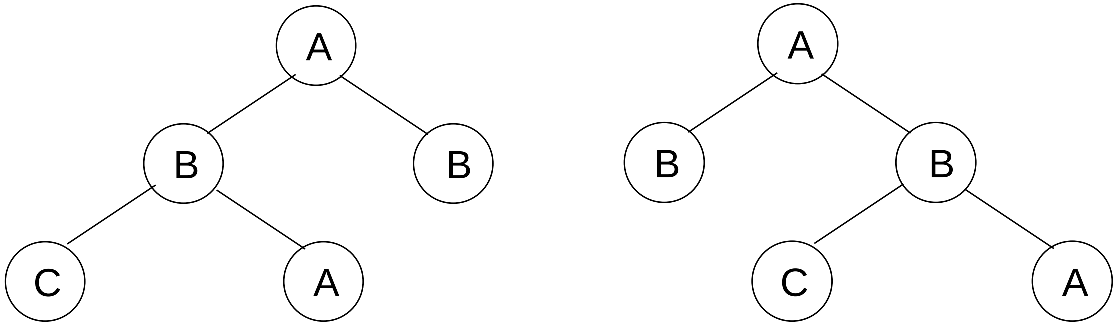
$$\phi_i\left(\mathcal{T}_1\right) = \sum_{n \in \mathcal{V}_1} I_i(n); \quad I_i(n) = \begin{cases} 1, & \text{sub-tree } i \text{ rooted at node } n \\ 0, & \text{otherwise} \end{cases}$$

- Can be made computationally efficient by recursively using a counting function

$$k(\mathcal{T}_1, \mathcal{T}_2) = \boldsymbol{\phi}(\mathcal{T}_1)^{\mathsf{T}} \boldsymbol{\phi}(\mathcal{T}_2) = \sum_{n_1 \in \mathcal{V}_1} \sum_{n_2 \in \mathcal{V}_2} f(n_1, n_2);$$

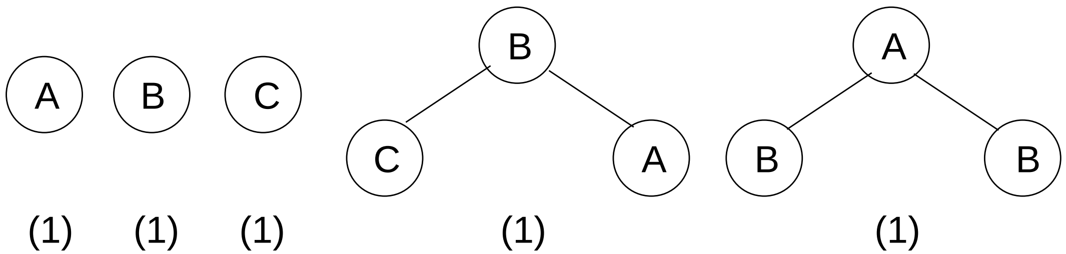
  - if productions from n_1 and n_2 differ $f(n_1, n_2) = 0$
  - for leaves $f(n_1, n_2) = \begin{cases} 1 & n_1 = n_2 \\ 0 & \text{otherwise} \end{cases}$
  - for non-leaf nodes $f(n_1, n_2) = \prod_{i=1}^{\#\text{ch}(n_1)} (1 + f(\text{ch}(n_1, i), \text{ch}(n_2, i)))$

Tree Kernel Example



Tree 1 ($\mathcal{T}_1$) Tree 2 ($\mathcal{T}_2$)

- The set of common sub-trees (and number) for these two graphs

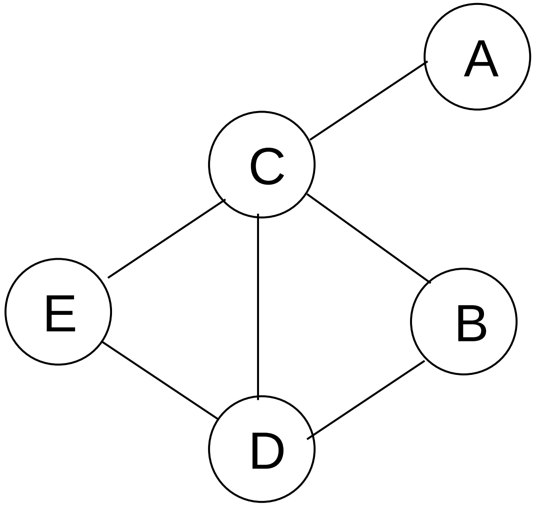


  - for these trees:

$$k(\mathcal{T}_1, \mathcal{T}_2) = 5$$

Graph Kernels

- An alternative form of kernel is based on graphs, $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$



- 5 nodes/vertices, $\mathcal{V} = \{A, B, C, D, E\}$, 6 edges, $\mathcal{E}$
- Various attributes:
  - adjacency matrix, $\boldsymbol{A}$: $a_{ij} = \begin{cases} 1, & (v_i, v_j) \in \mathcal{E} \\ 0, & \text{otherwise} \end{cases}$
  - walk length $k{-}1$, $w = \{v_1, \ldots, v_k\}$, $(v_{i-1}, v_i) \in \mathcal{E}$
  - edges may also have weights associated with it
- Walks of length k can be computed using $\boldsymbol{A}^k$

- For the example graph above

$$\boldsymbol{A} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \boldsymbol{A}^2 = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 2 & 1 & 1 & 2 \\ 0 & 1 & 4 & 2 & 1 \\ 1 & 1 & 2 & 3 & 1 \\ 1 & 2 & 1 & 1 & 2 \end{bmatrix} \quad \boldsymbol{A}^3 = \begin{bmatrix} 0 & 1 & 4 & 2 & 1 \\ 1 & 2 & 6 & 5 & 2 \\ 4 & 6 & 4 & 6 & 6 \\ 2 & 5 & 6 & 4 & 5 \\ 1 & 2 & 6 & 5 & 2 \end{bmatrix}$$

Graph Kernels

How close are two graphs, $\mathcal{G}_1$ and $\mathcal{G}_2$ to each other?

- Set of kernels that operate on these graphs - $k(\mathcal{G}_1, \mathcal{G}_2)$
  - based on common paths/walks in the two graphs
  - could consider longest/shortest paths
- Random walk kernel counts the number of matching walks in the two graphs
  - based in the product graph of $\mathcal{G}_1$ and $\mathcal{G}_2$, $\mathcal{G}_\text{x}$
    $\mathcal{G}_\text{x}$ graph of all identically labelled nodes and edges from $\mathcal{G}_1$ and $\mathcal{G}_2$

$$k(\mathcal{G}_1, \mathcal{G}_2) = \sum_{i,j=1}^{|\mathcal{V}_\text{x}|} \left[\sum_{n=0}^{\infty} \lambda^n \boldsymbol{A}_\text{x}^n / n! \right]_{ij} = \sum_{i,j=1}^{|\mathcal{V}_\text{x}|} \left[\exp\left(\lambda \boldsymbol{A}_\text{x}\right) \right]_{ij}$$

  - $\boldsymbol{A}_\text{x}$ is the adjacency matrix for the product graph $\mathcal{G}_\text{x}$
  - λ is a scalar to weight the contribution of longer walks

Perceptron Algorithm

- It is possible to use kernel functions on other classifiers
- Consider the perceptron algorithm (lecture 2). which can be written as

```
Initialise w = 0, k = 0 and b = 0;
Until all points correctly classified do:
      k=k+1;
      if x_k is misclassified then
            w = w + y_k x_k
            b = b + y_k
```

  – this yields the linear decision boundary defined by $\mathbf{w}, b$

- Classification based on

$$g(\boldsymbol{x}) = \mathbf{w}^{\mathsf{T}}\boldsymbol{x} + b, \quad \hat{\omega} = \begin{cases} \omega_1, & \text{if } g(\boldsymbol{x}) > 0 \\ \omega_2, & \text{otherwise} \end{cases}$$

Kernelised Perceptron Algorithm

- The kernelised version of the algorithm may be described as

```
Initialise α_i = 0, i = 1,...,n, k = 0 and b = 0;
Until all points correctly classified do:
      k=k+1;
      if x_k is misclassified then
            α_k = α_k + 1
            b = b + y_k
```

  – "Lagrange multiplier", α_i, the number of times sample $\boldsymbol{x}_i$ is mis-recognised

- Classification is then performed based on (as for the SVM)

$$g(\boldsymbol{x}) = \sum_{i=1}^{n} y_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}_i) + b, \quad \hat{\omega} = \begin{cases} \omega_1, & \text{if } g(\boldsymbol{x}) > 0 \\ \omega_2, & \text{otherwise} \end{cases}$$