

# **4F10: Deep Learning and Structured Data**

## **Kernels for Structured Data**

**José Miguel Hernández-Lobato**

Department of Engineering  
University of Cambridge

Michaelmas Term

# Motivation

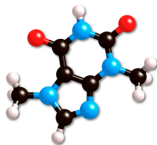
The kernels that we have seen so far only work with fixed length input vectors.

However, many real world data sets are often non-vectorial!

Biological sequences



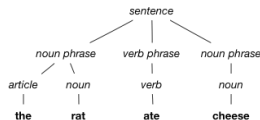
Molecules



Text documents

signal.  
nary code with which the present  
ls may take various forms, all of  
e property that the symbol (or  
representing each number (or signi  
differs from the ones representi  
er and the next higher number (o  
litude) in only one digit (or puls  
Because this code in its primary  
built up from the conventional  
a sort of reflection process and l  
rms may in turn be built up fro  
form in similar fashion, the c  
, which has as yet no recognized  
anted in this specification and  
s the "reflected binary code."  
a receiver station, reflected bina

Parse trees



How to apply machine learning methods in these cases?

**Possible solution:** use kernels for strings, trees and graphs!

**Two step general approach:**

- 1 Extract features from input objects.
- 2 Compute kernels (dot products) using those features.

Specific algorithms can be used to obtain computationally efficient implementations.

# String kernel

Each data point  $x$  is a string of characters from alphabet  $\mathcal{A}$ , that is,  $x \in \mathcal{A}^*$ , where  $*$  denotes the Kleene closure operation.

## Main idea:

- 1 Given list of substrings  $s_1, s_2, s_3, \dots \in \mathcal{A}^*$ , each string  $x$  is encoded using the feature vector  $\phi(x) = (\phi_{s_1}(x), \phi_{s_2}(x), \phi_{s_3}(x), \dots)^T$ .
- 2 For each substring  $s$ , the feature  $\phi_s(x)$  indicates the occurrence of  $s$  in  $x$ :  
 $\phi_s(x) = 0$  if  $s$  does not occur in  $x$  and  $\phi_s(x) > 0$ , otherwise.
- 3 The kernel between two strings  $x$  and  $x'$  is defined as the dot product

$$k(x, x') = \phi(x)^T \phi(x') = \sum_{s \in \{s_1, s_2, s_3, \dots\}} \phi_s(x) \phi_s(x').$$

## Example: gap-weighted kernel

Substring  $s$  may not need to be continuous in  $x$ , but the more gaps in the occurrence, the smaller the value of  $\phi_s(x)$ . For  $n$  gaps we could have  $\phi_s(x) = \lambda^n$ , where  $0 < \lambda < 1$ .

For the list of substrings  $ca, ct, cr, ar, rt, ba, br$ :

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

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

Can be computed efficiently for **all possible substrings** using dynamic programming.

## Another example: $k$ -spectrum kernel

The  $k$ -spectrum kernel considers all possible subsequences of length  $k$ .

$\phi_s(x)$  is in this case defined as the number of times that  $s$  **exactly** occurs in  $x$ .

$k(x, x')$  computed fast by using **suffix tree** to store all length- $k$  sequences in  $x$  and  $x'$ .

**Example:** classification of DNA sequences.

For the case  $k = 3$ :

$x$  **AAACAAATAAGTAACTAATCTTTTAGGAAGACGTTTCAACCATTTTGAG**

$x'$  **TACCTAATTATGAAATTAAATTTTCAGTGTGCTGATGGAAACGGAGAAGTC**

$s$	AAA	AAC	...	CCA	CCC	...	TTT
# in $x$	2	4	...	1	0	...	3
# in $x'$	3	1	...	0	0	...	1

$$k(x, x') = 2 \cdot 3 + 4 \cdot 1 + \dots 1 \cdot 0 + 0 \cdot 0 \dots 3 \cdot 1$$

Co-occurrence of substrings is more informative for long substrings  $\rightarrow$   **$k$  should be large**.  
However, common occurrences decrease as  $k$  increases  $\rightarrow$   **$k$  should be small**.

**Bag-of-words kernel** obtained when  $k = 1$ . Often applied to documents (word strings).

Slide source: Gunnar Rätsch

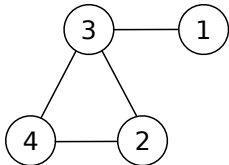
Classification of DNA sequences

Kernel	auROC
Linear	88.2%
Polynomial $d = 7$	90.4%
Spectrum $k = 1$	94.0%
Spectrum $k = 3$	96.4%
Spectrum $k = 5$	94.5%

Linear and polynomial kernels use counts of only a few informative sub-sequences.

# Graphs and graph walks

Each data point is a graph  $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$  with  $\mathcal{E} = \{(i, j) : i, j \in \mathcal{V}\}$ .



- The node set is  $\mathcal{V} = \{1, 2, 3, 4\}$ , and the edge set is  $\mathcal{E} = \{(1, 3), (3, 2), (3, 4), (2, 4)\}$ .
- The  $|\mathcal{V}| \times |\mathcal{V}|$  adjacency matrix  $A$  satisfies  $a_{i,j} = 1$  if  $(i, j) \in \mathcal{E}$  and  $a_{i,j} = 0$ , otherwise.

## Graph walks:

A  $k$ -length walk is defined as  $w = \{v_1, \dots, v_{k+1}\}$ , where  $(v_i, v_{i+1}) \in \mathcal{E}$ .

The number of  $k$ -length walks between nodes  $i$  and  $j$  is given by  $[A^k]_{i,j}$  since

$$[A^2]_{i,j} = \sum_{s_1=1}^{|\mathcal{V}|} a_{i,s_1} a_{s_1,j}, \quad [A^k]_{i,j} = \sum_{s_1=1}^{|\mathcal{V}|} \cdots \sum_{s_{k-1}=1}^{|\mathcal{V}|} a_{i,s_1} a_{s_1,s_2} \cdots a_{s_{k-1},j}.$$

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ 0 & 1 & 3 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}, \quad A^3 = \begin{bmatrix} 0 & 1 & 3 & 1 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 2 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}.$$

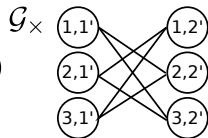
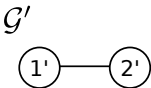
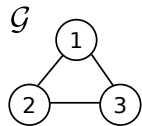
**Random-walk graph kernel:** the  $i$ -th entry in  $\phi(\mathcal{G})$  counts the number of  $i$ -length walks in  $\mathcal{G}$ .

# Random-walk graph kernel

$k(\mathcal{G}, \mathcal{G}')$  counts the number of **common walks in graphs**  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  and  $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ .

Can be computed efficiently using the **direct product graph** of  $\mathcal{G}$  and  $\mathcal{G}'$ :

$$\mathcal{G}_\times = (\mathcal{V}_\times, \mathcal{E}_\times) \quad \text{where} \quad \begin{aligned} \mathcal{V}_\times &= \{(a, b) : a \in \mathcal{V} \text{ and } b \in \mathcal{V}'\} \\ \mathcal{E}_\times &= \{((a, a'), (b, b')) : (a, b) \in \mathcal{E} \text{ and } (a', b') \in \mathcal{E}'\} \end{aligned}$$



Each walk in  $\mathcal{G}_\times$  corresponds to one walk in  $\mathcal{G}$  and  $\mathcal{G}'$ .

$\mathcal{G}_\times$  satisfies  $A_\times = A \otimes A'$ , where  $\otimes$  is the **Kronecker product**.

Assuming that  $\lambda > 0$  is small enough to guarantee convergence, the kernel is given by

$$k(\mathcal{G}, \mathcal{G}') = \sum_{i,j=1}^{|\mathcal{V}_\times|} \left[ \sum_{n=0}^{\infty} \lambda^n A_\times^n \right]_{i,j} = \mathbf{1}^T \underbrace{[\mathbf{I} - \lambda A \otimes A']^{-1}}_{\mathbf{x}} \mathbf{1} \Leftrightarrow \mathbf{x} = \mathbf{1} + \lambda (A \otimes A') \mathbf{x},$$

**Efficient evaluation of the kernel** (only for reference):

$\mathbf{x}$  found by iterating  $\mathbf{x}_{t+1} = \mathbf{1} + \lambda (A \otimes A') \mathbf{x}_t$ , where  $(A \otimes A') \mathbf{x}_t = \text{vec}(A' \text{vec}^{-1}(\mathbf{x}_t) A^T)$ ,  $\text{vec}(\cdot)$  vectorises a matrix by stacking columns and  $\text{vec}^{-1}(\cdot)$  is its inverse operation.

**Problems:** a walk can visit the same cycle again and again  $\rightarrow$  small structural similarities can produce huge kernel values. High cost:  $\mathcal{O}(n^3)$  for  $n \times n$  matrices  $A$  and  $A'$ .

# Weisfeiler-Lehman graph kernel

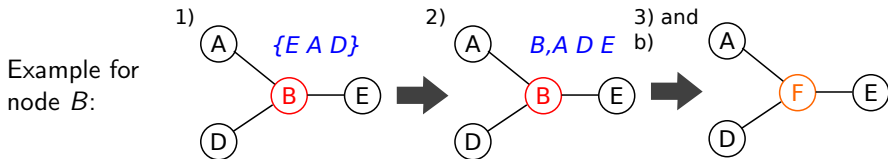
Lower cost (can work with **large graphs**) and allows for **node labels**.

Repeat, for  $M$  iterations, for each graph in the dataset:

a) Repeat for each node in the current graph:

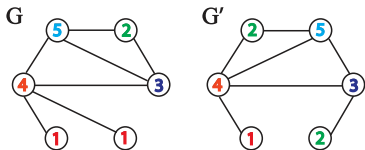
- 1 Create a set with labels of adjacent vertices (for  $B$  we get  $\{EAD\}$ ).
- 2 Sort the label set and add vertex label as a prefix ( $B, ADE$ ).
- 3 Compress resulting label sequence into **unique value** ( $B, ADE \rightarrow F$ ).

b) Assign to each graph node its resulting **unique value** as new **vertex label**.

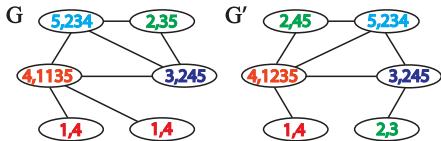


The graph kernel is finally obtained by applying the **bag-of-words kernel** to the **vertex labels** obtained throughout all the iterations (including the initial labels).

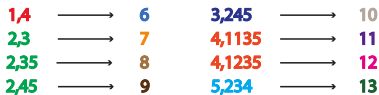
# One-iteration example and results on molecules



Result of steps 1 and 2: multiset-label determination and sorting



Result of step 3: label compression



Features through bag-of-words kernel:

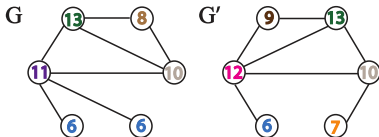
$$\phi^{(1)}(G) = (\mathbf{2, 1, 1, 1, 1, 2, 0, 1, 0, 1, 1, 0, 1})^T$$

$$\phi^{(1)}(G') = (\mathbf{1, 2, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1})^T$$

Counts of  
original  
node labels

Counts of  
compressed  
node labels

Result of step 4: relabeling



Kernel:

$$k^{(1)}(G, G') = \phi^{(1)}(G)^T \phi^{(1)}(G') = 11.$$

Classification accuracy on molecule data:

Method/Data Set	MUTAG	NC11	NC109
WL	82.05 (±0.36)	82.19 (±0.18)	82.46 (±0.24)
Random walk	80.72 (±0.38)	64.34 (±0.27)	63.51 (±0.18)

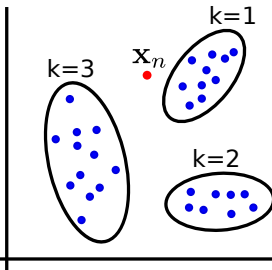


# Fisher kernel

**Main idea:** use a **probabilistic generative model** to obtain a **fixed-length vector representation** of complex structured data.

- 1 Train generative model  $p(x|\theta)$  on available input data  $x_1, \dots, x_N$ , e.g. by MLE.
- 2 Define the Fisher score vector as  $\phi(x_n) = \nabla_{\theta} \log p(x_n|\theta)|_{\theta_{MLE}}$ .
- 3 The **naive Fisher kernel** is given by  $k(x_n, x_m) = \phi(x_n)^T \phi(x_m)$ .

**Example:** let us assume that the generative model is a **mixture model**, that is,  $p(x|\theta) = \sum_{k=1}^K p(x|\theta_k)\pi_k$ . Then, the component of  $\phi(x_n)$  associated with  $\pi_k$  is


$$[\phi(x_n)]_{\pi_k} = \left. \frac{\partial \log p(x|\theta)}{\partial \pi_k} \right|_{\theta=\theta_{MLE}} = \frac{p(x|\theta_k^{MLE})}{\sum_k p(x|\theta_k^{MLE})\pi_k^{MLE}},$$

which measures the amount by which the  $k$ -th component contributes to generate  $x_n$ .

Thus,  $\phi(x_n)$  separates input space into regions according to relative weight of mixture components.

In the example from the figure, we would have

$$[\phi(x_n)]_{\pi_1} = 2.8 \quad , \quad [\phi(x_n)]_{\pi_2} = 0.01 \quad , \quad [\phi(x_n)]_{\pi_3} = 0.09$$

# Summary

Kernels can be defined on structured data such as text, trees, graphs, molecules, etc...

They 1) extract features from data and 2) compute dot products using those features.

Specific algorithms can be used for efficient computation of dot products.

Gap-weighted,  $k$ -spectrum and bag-of-words are types of string kernels.

Tree kernels can be obtained by counting common subset trees using recursion.

Graph kernels can be implemented by counting common walks, but this has a high cost.

Weisfeiler-Lehman graph kernel has lower cost and can perform better in practice.

Fisher kernel uses generative model to map structured data to fix-length score vectors.