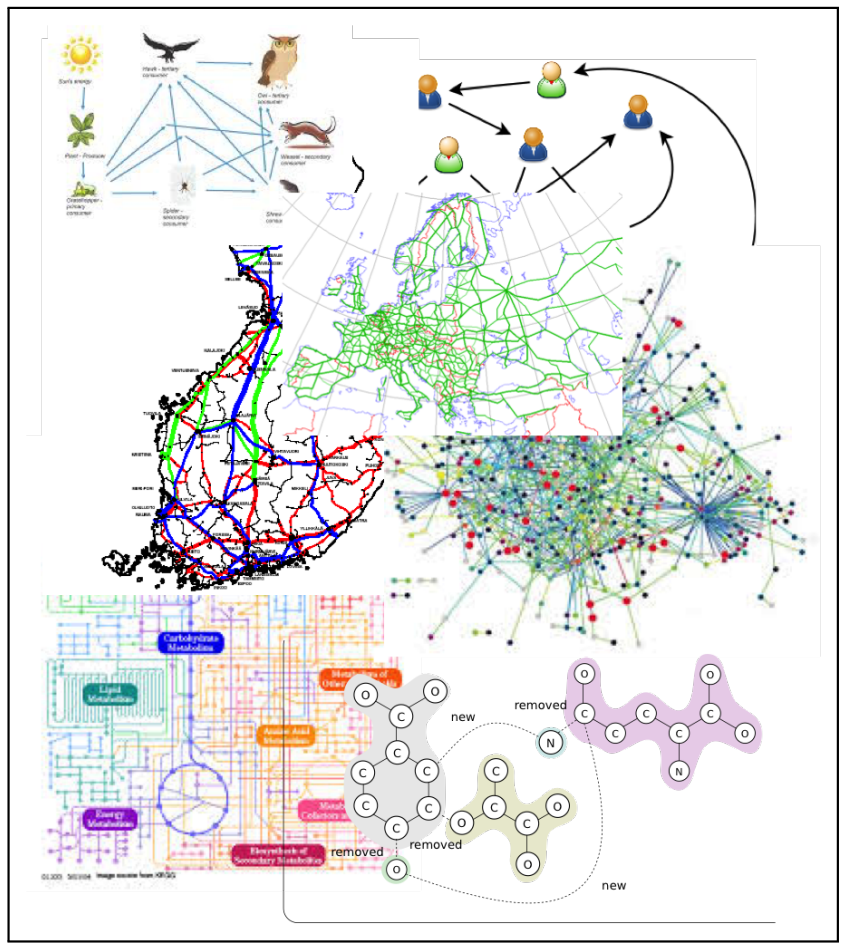
CS-E4830 Kernel Methods in Machine Learning

Lecture 9: Learning on graphs

15. November, 2017

"Graphs are everywhere"

- Graphs (or networks) are found in many fields of life:
  - Computer networks, internet: computers + communication channels
  - Social networks: "people" + "friendship"
  - Bioinformatics, protein interaction networks: protein molecules as nodes, physical binding as edges
  - Drug discovery: atom species as nodes, bonds as edges
- Making use of the graph structure inherent to a data dource maybe crucial for success of machine learning



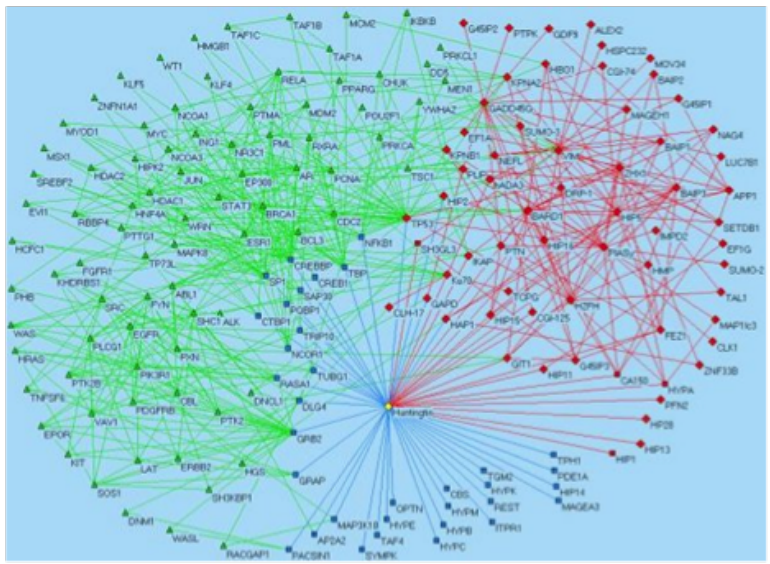
Machine learning on Graphs

Machine learning teaks on graphs:

- Graph labeling: Given a graph, predict labels for its nodes
  - Typically nodes correspond to examples, the edges to similarity between examples
  - c.f. Semi-supervised learning with Graph Laplacians
- Link Prediction: Given a pair of nodes, predict if they should be connected with an edge
  - Typically nodes correspond to examples, the edges to interaction between examples
- Graph classification: given a graph decide which class is belongs
  - Nodes correspond to variables, the edges to interaction between variables

Link prediction as a pattern recognition problem[1]

- Assume a set of nodes $V = \{v_1, \ldots, v_n\}$ corresponding to the entity of interest (people, documents, genes, ...)
- Each node has an associated feature vector $\phi(v)$ describing its properties, behaviour, history, etc. (e.g. person's likes/don't likes, keywords of the document, expression levels of a gene)
- We wish to reconstruct a set of edges $E \subset V \times V$ that define the network



[1] Vert, 2008

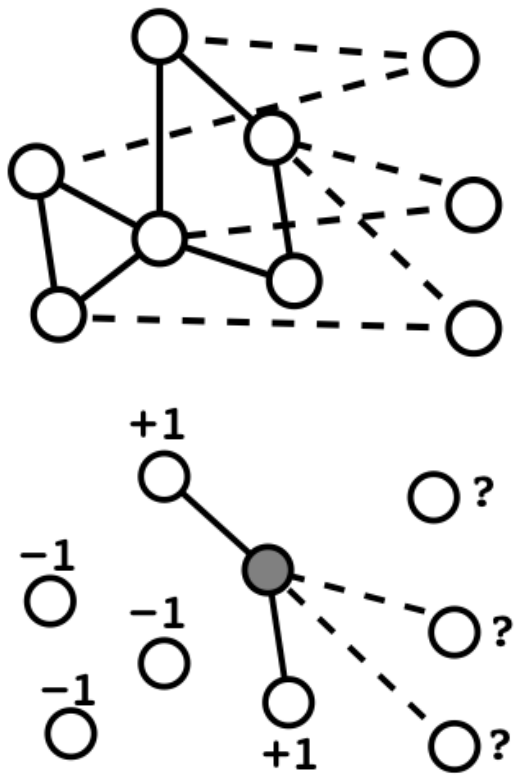
de novo link prediction vs. graph completion

- **de novo link prediction** would entail predicting the set of edges E from the feature vectors of the nodes alone
  - This is very hard computationally and statistically, requires huge numbers of samples and very small networks to succeed
- In many cases, part of the network is typically "known" already but the *de novo* approach does not make use of that information!
- Instead, we will assume that part of the network is already known, and our task is to **complete the network** by filling in the missing edges
  - Network/graph completion is potentially an easier task than reconstructing the whole graph
  - Conforms better to real world applications (e.g. "people you might know" in social networks)

Global and local models

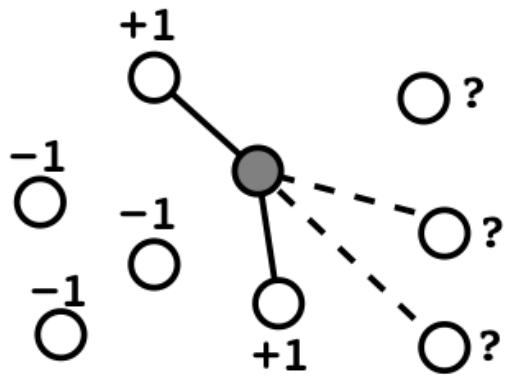
The graph/network completion problem can be solved by global or local models

- A **global model is** trained to predict the absense or presence of any edge in the network, a single model is needed
- A **local model** predicts the edges adjacent to a *seed* node, need one model per node:
  - Translates the local problem into a classification problem, can use standard classifiers such as SVM
- In both cases the known edges are used to construct a training set from which a predictive model is learned



Link prediction with local models

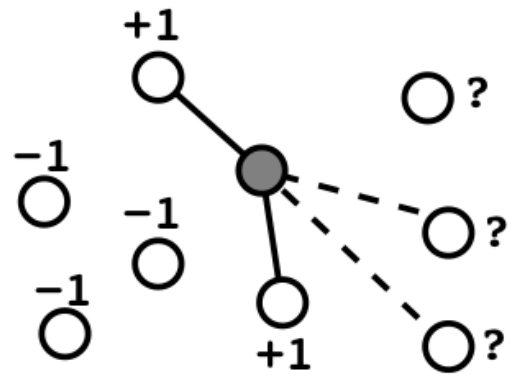
- The task is to predict presence/absence of edges between a given center node v and the other nodes.
- Assume each node v is described by a feature vector $\phi(v)$ (e.g. profile data of a social network user v)
- Denote by $V_v \subset V$ the set of nodes $u \in V$ for which we know whether (v, u) exists or not



Link prediction with local models

Repeat for all nodes in the graph:

- Create a local training set $S_v = \{(\phi(u_i), y_i) | u_i \in V_v)\}$, where for the edge $e = (v, u_i)$ the label indicates if the edge is present ($y_i = 1$) or absent ($y_i = -1$)
- Train SVM with the local training set of node v to obtain a score: $f_v(u) = \langle \mathbf{w}_v, \phi(u) \rangle + b_v$
- Predict with SVM the label y_j for each pair (v, u_j) where the edge label is not known



Obtaining negative examples

- For binary classification we need knowledge about edges that are **known** to be absent
- This is challenging as typically most of data available is positive data: interactions that are known to be present
- We need to generate **pseudo-negative examples**: take random pairs of nodes that are **not known to be** connected and declare them **not to be** connected
  - Rationale: typical networks are sparse, only small fraction of nodes are connected, so a randomly drawn pair rarely is connected.
  - Chance of introducing errors to the network, especially if we generate lots of negative examples
  - We can use background knowledge to choose negative examples in order to decrease this chance

Use for undirected graphs

- The approach is directly applicable for directed graphs.
- For undirected graphs, each undirected node pair $\{v, u\}$ in the training set should be considered twice, once in each direction.
- To extract the prediction for an undirected edge, the two directed predictions should be combined e.g. by averaging the scores:

$$f(\{u, v\}) = (f_v(u) + f_u(v))/2,$$

  where we denote by $f_v(u)$ the score for positive edge label for edge (v, u) of local classifier at node v.
- If average score is positive predict an edge $\{u, v\}$.

Pros and cons of the local method

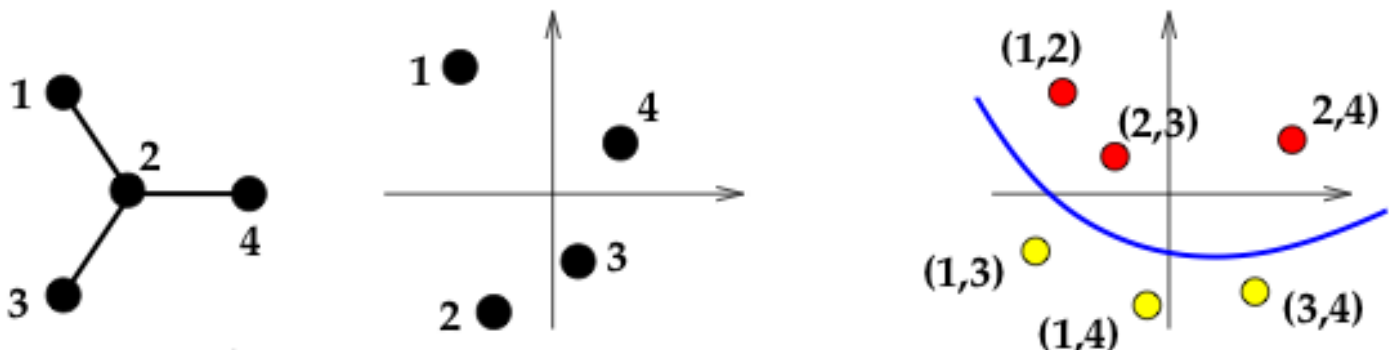
- Splitting a large network problem into a set of local problems can be beneficial in terms of computation time
  - Time to train and predict in each node gets smaller
  - Parallel architectures can be easily used as the local problems are treated independent
- Data fragmentation is a potential pitfall: if there are not enough examples for some seed nodes, accuracy of the model can suffer

Pros and cons of the local method

- Local approach splits the data into independent units
- Information sharing between the local problems is not possible
  - e.g. if (u, v) interact, u is similar to u' and v is similar to v', the pair (u', v') is likely to interact a well
- The local approach only uses pairs with a single node as the center, so this information is not used
- To make use of the above kind of information, the model needs to be defined on edges (or pairs of nodes), not single nodes

Link prediction with global models

- The ingredients we have: the known edges of the graph (left picture), features of the nodes $\phi(v)$ (middle picture)
- Goal: We wish to represent each pair of nodes by a **joint feature vector** $\psi(u, v)$ which should contain features predictive of the interaction of that pair
- Using this representation the classifier then learns to separate interacting pairs from non-interacting pairs (right picture)



Features for pairs of nodes

- Consider building a joint feature vector $\boldsymbol{\psi}(u, v)$ for a pair of nodes (u, v) from feature vectors of the nodes $\boldsymbol{\phi}(u), \boldsymbol{\phi}(v)$
- We generally want to enable learning from correlations of node features (e.g. $\phi_i(u)$ and $\phi_j(v)$ are 'high' at the same time)
  - We do not assume that exactly the same features correlate i.e. we may have $i \neq j$
  - e.g. interaction might require a lock-and-key property, with one object possessing the 'lock features' and the other 'key features'
- To build all feature pairs we take the **tensor product**, which contains the products of all feature pairs:

$$\psi(u, v)_k = \phi_i(u)\phi_j(v), \text{ where } k = N \cdot (j - 1) + i$$

$$\boldsymbol{\psi}_{\times}(u, v) = \boldsymbol{\phi}(u) \otimes \boldsymbol{\phi}(v) = \left(\psi(u, v)_k\right)_{k=1}^{N^2}$$

Tensor product of feature vectors

Given two vectors, $\phi_1 \in \mathbb{R}^{N_1}$ and $\phi_2 \in \mathbb{R}^{N_2}$, their (also called outer product or direct product) $\psi_\times = \phi_1 \otimes \phi_2$ can be alternatively represented by

- A matrix $\phi_1 \phi_2{}' \in \mathbb{R}^{N_1 \times N_2}$ with entries

$$\Psi_\times = \begin{bmatrix} \phi_{1,1}\phi_{2,1} & \cdots & \phi_{1,1}\phi_{2,j} & \cdots & \phi_{11}\phi_{2,N_2} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \phi_{1,i}\phi_{2,1} & \cdots & \phi_{1,i}\phi_{2,j} & \cdots & \phi_{1i}\phi_{2,N_2} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \phi_{1,N_1}\phi_{2,1} & \cdots & \phi_{1,N_1}\phi_{2,j} & \cdots & \phi_{1,N_1}\phi_{2,N_2} \end{bmatrix}$$

- A stacked vector of columns of $\Psi_\times$, $\psi_\times \in \mathbb{R}^{N_1 N_2} = \mathbf{vec}(\Psi_\times) = (\phi_1{}' \cdot \phi_{21}, \ldots, \phi_1{}' \cdot \phi_{2j}, \ldots, \phi_1{}' \cdot \phi_{2,N_1})'$
- A stacked vector of rows of $\Psi_\times$, $\hat{\psi}_\times \in \mathbb{R}^{N_1 N_2} = \mathbf{vec}(\Psi_\times{}') = (\phi_{11} \cdot \phi_2{}', \ldots, \phi_{1i} \cdot \phi_2{}', \ldots, \phi_{1,N_1} \cdot \phi_2{}')'$
- All representations contain the same elements, indexed differently: $(\Psi_\times)_{ij} = \psi_{\times,k} = \hat{\psi}_{\times,h}$, where $k = N_1 \cdot (j-1) + i$ and $h = N_2 \cdot (i-1) + j$

Kernels of tensor products

Depending on the representation, the **same** tensor product kernel can be computed in different ways:

- Frobenius inner product of two matrices:
  $\kappa_\times(x,z) = \langle \Psi_\times(x), \Psi_\times(z) \rangle_F = \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} \Psi_\times(x)_{ij} \Psi_\times(z)_{ij}$
- Inner product of stacked column vectors:
  $\kappa_\times(x,z) = \mathbf{vec}(\Psi_\times(x))' \mathbf{vec}(\Psi_\times(z))$
- Inner product of stacked row vectors:
  $\kappa_\times(x,z) = \mathbf{vec}(\Psi_\times(x)')' \mathbf{vec}(\Psi_\times(z)')$
- Itemwise product of kernels: $\kappa_\times(x,z) = \kappa_1(x,z) \cdot \kappa_2(x,z)$, where $\kappa_1(x,z) = \phi_1(x)'\phi_1(z)$ and $\kappa_2(x,z) = \phi_2(x)'\phi_2(z)$

Tensor product pairwise kernel (TPPK)

- The tensor product feature representation $\psi_{\times}(u, v)$ is dependent on the order of u and v: in general $\psi_{\times}(u, v) \neq \psi_{\times}(v, u)$
- Suitable directly to model directed edges, but less good for undirected edges
- For undirected graphs one can add the directed features

$$\psi_{\times}(\{u, v\}) = \psi_{\times}(u, v) + \psi_{\times}(v, u)$$

- TPPK kernel can be computed as combination of the node kernels as follows

$$\begin{aligned}\kappa_{TPPK}(\{a, b\}, \{c, d\}) &= \psi_{\times}(\{a, b\})' \psi_{\times}(\{c, d\}) = \\ &= \kappa_V(a, c) \cdot \kappa_V(b, d) + \kappa_V(a, d) \cdot \kappa_V(b, c) + \\ &\quad \kappa_V(b, c) \cdot \kappa_V(a, d) + \kappa_V(b, d) \cdot \kappa_V(a, c)\end{aligned}$$

where $\kappa_V(u, v) = \phi(u)'\phi(v)$ is the kernel similarity of nodes.

Putting it together

- With the global model, the training and prediction setup is straight-forward
- We take the training set of labeled edges $\mathcal{S} = \{((u_1, v_1), y_1), \ldots, ((u_\ell, v_\ell), y_\ell)\}$
- Construct the TPPK kernel: $\kappa_{TPPK}(\{u_i, v_i\}, \{u_j, v_j\})$ for all $i, j = 1, \ldots, \ell$
- Train a single SVM model using the kernel and the labels
- For each pair not in the training set, the edge score is obtained from

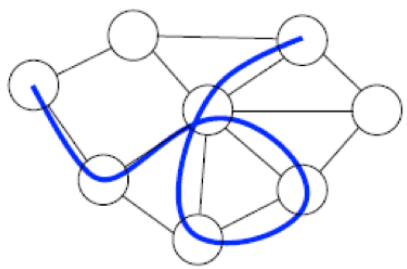
$$f(\{u, v\}) = \sum_{i=1}^{\ell} \alpha_i y_i \kappa_{TPPK}(\{u_i, v_i\}, \{u, v\})$$

Graph classification

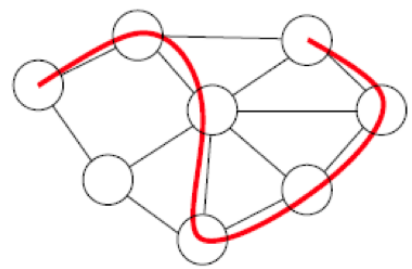
- Task: given a graph predict which class it belongs
- Training Data: $\{(g_i, y_i\}_{i=1}^{\ell}$ where $g_i = (V_i, E_i)$ is a graph, associated with a class $y_i \in Y$ (can be binary or multiclass)
- **Difference to link prediction**: each data point is a graph, with its own nodes and edges, target variable $y \in Y$ concerns the graph as a whole.
- Many applications, especially in bioinformatics, for example
  - Drug discovery: given a candidate drug molecule (graph), predict if it will be active against a given type of cancer cell
  - Protein function prediction: given a 3D protein structure, predict its functional role

Definitions: Labeled graphs, walks and paths

- A **graph** G is a set of nodes (or nodes) V and edges E, where $E \subset V^2$.
- A **labeled graph** is a graph with labels on nodes and/or edges, labels can be
  - discrete or continuous valued
  - scalars or vector-valued (similarly to factored string kernel)
- A **walk** w of length $k-1$ in a graph is a sequence of nodes $w = (v_1, v_2, \cdots, v_k)$ where $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$.
- w is a **path** if $v_i \neq v_j$ for $i \neq j$, i.e. a walk with no cycles



walks



Paths

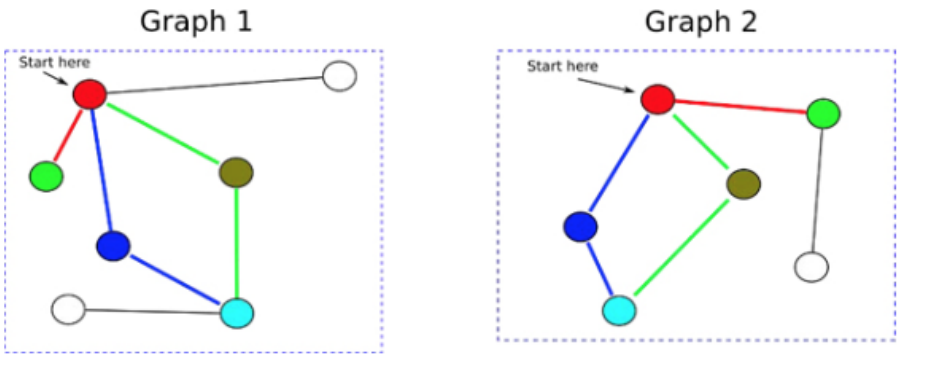
Graph kernels

- Basic idea: count common substructures in two graphs
- Example substructures:
  - Walks
  - Paths
  - Cyclic patterns
  - Tree-shaped subgraphs
  - (small) General subgraphs
- Generally:
  - Huge sparse feature spaces, exponential in the size of the graphs $\Longrightarrow$ we will not want to represent features explicitly
  - Trade-off between computational complexity and accuracy, need to avoid NP-hard problems but still use informative features
  - Polynomial-time kernel computation possible for selected feature representations

Graph kernels based on random walks

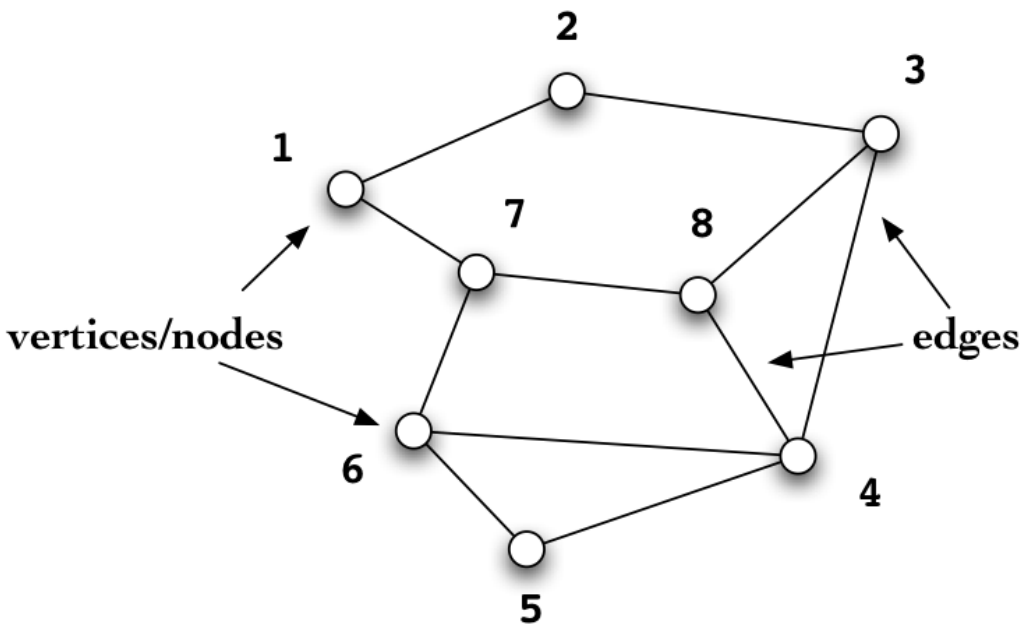
Walk kernel idea:

- Count the number of matching walks in two graphs
- In an unlabeled graph two walks match if they have the same length
- In a labeled graph also the node and edge labels need to match



From adjacency matrix to walks

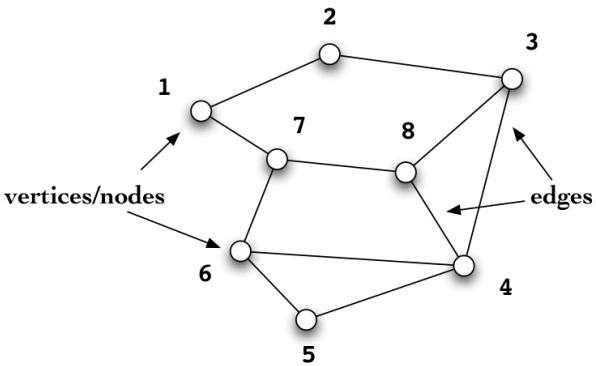
- To compute number of walks in a **single unlabeled graph**, we can use the adjacency matrix A of the graph
- $A(i,j) = \begin{cases} 1, & \text{if there is an edge from node } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$
- An edge (v_i, v_j) is a length-1 walk $\implies$ $A(i,j) =$ number of length-1 walks from v_i to v_j



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

From adjacency matrix to walks

- Matrix multiplication $A^2 = AA$ reveals the number of length 2 walks $A^2_{ij} = \sum_h A_{ih} A_{hj}$



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

- Further multiplication combines the counts of length $k-1$ walks from v_i to v_h with a length 1 walk from v_h to v_j, for all intermediary nodes v_h

$$A^k_{ij} = \sum_h A^{k-1}_{ih} A_{hj}$$

- $S = \sum_{n=0}^{\infty} \lambda^n A^n_{\times}$, S_{ij} counts a weighted sum of walks of all lengths between nodes i and j, decay $0 < \lambda < 1$ keeps the sum finite

Product Graph: counting common walks

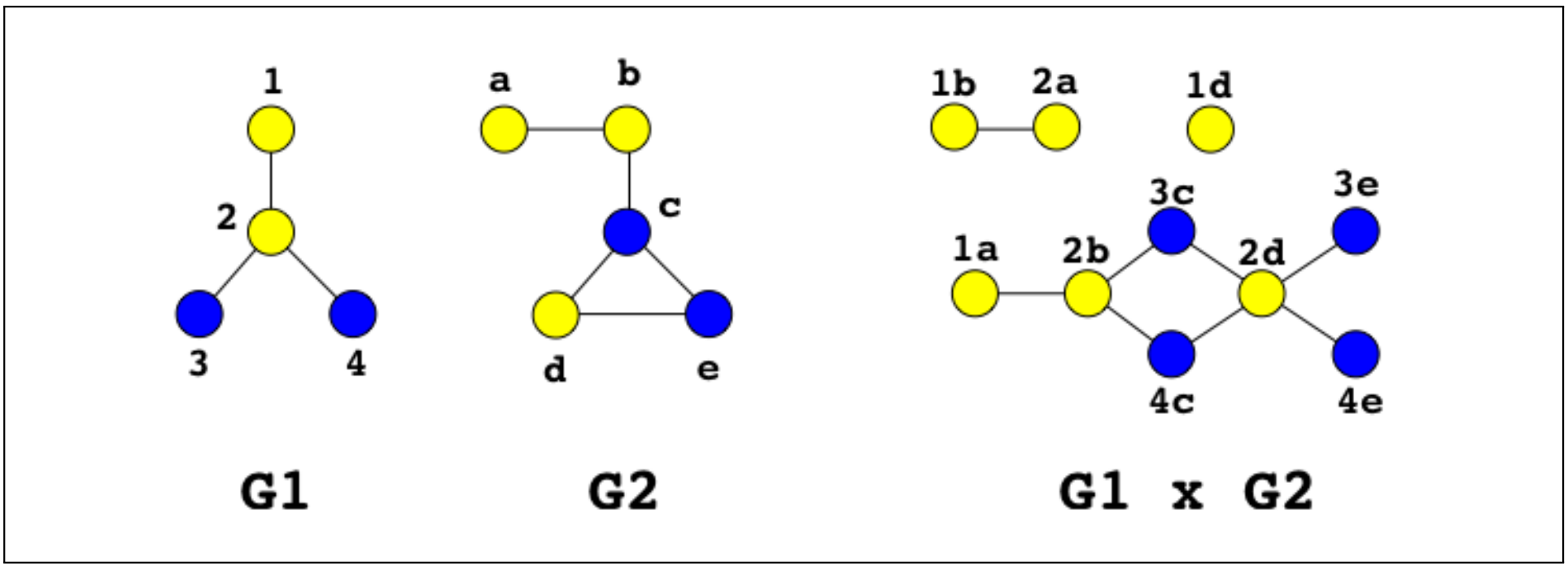
- Compute the walk kernel, we need to count common walks in two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$
- For this we can use the product graph $G_\times = (V_\times, E_\times)$, given by the following node and edge sets

$$
\begin{aligned}
V_\times = &\{(v_1, v_2) | v_1 \in V_1, v_2 \in V_2, \\
&\quad label(v_1) = label(v_2)\} \\
E_\times = &\{((u_1, v_1), (u_2, v_2)) \in E_1 \times E_2) \,|(u_1, u_2) \in V_\times, (v_1, v_2) \in V_\times)) \\
&\quad label((u_1, v_1)) = label((u_2, v_2))\}
\end{aligned}
$$

- Product graph consists all identically labeled node pairs from the two graphs, connected with edges
- An edge $e_\times = (e_1, e_2) = ((u_1, v_1), (u_2, v_2))$ occurs in a product graph when both original graphs have the corresponding identically labeled edges (u_1, v_1) and (u_2, v_2)

Product graph

- Tracing a walk in the product graph corresponds to simultaneously tracing common walks in the two original graphs



Random Walk Kernel

- Common **labeled** walks of length k can now be computed from the adjacency matrix of the product graph $A_\times^k$
  - Note a trick: The labels in the product graph can be ignored, and we can treat the graph as unabeled
  - The structure of the product graph ensures that only walk with matching labels are counted
- The random walk kernel is given by

$$\kappa_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda^n A_\times^n]_{ij},$$

- Random walk kernel counts all pairs of matching walks of any length (including dummy walks of length 0 through $\lambda^0 A^0 = I_{|V_\times|}$)
- $0 < \lambda < 1$ is decaying factor for the sum to converge
- The underlying feature space is now infinite-dimensional!

Computing the Random walk kernel

- The kernel can be expressed

$$\kappa_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda^n A_\times^n]_{ij} = \sum_{i,j=1}^{|V_\times|} s_{ij}$$

  where $S = (s_{ij})_{i,j=1}^{|V_\times|} = \sum_{n=0}^{\infty} \lambda^n A_\times^n$ is a geometric matrix series
- Multiply the series $S = \sum_{n=0}^{\infty} \lambda^n A_\times^n = I_{|V_\times|} + \lambda A_\times + \lambda^2 A_\times^2 + \cdots$ by $\lambda A_\times$ to obtain

$$\lambda A_\times S = \lambda A_\times + \lambda^2 A_\times^2 + \cdots = \sum_{n=1}^{\infty} \lambda^n A_\times^n = S - I_{|V_\times|}$$

- Solve the equation $\lambda A_\times S = S - I_{|V_\times|}$ for $S \implies S = (I_{|V_\times|} - \lambda A_\times)^{-1}$
- Thus the kernel is given by

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [(I_{|V_\times|} - \lambda A_\times)^{-1}]$$

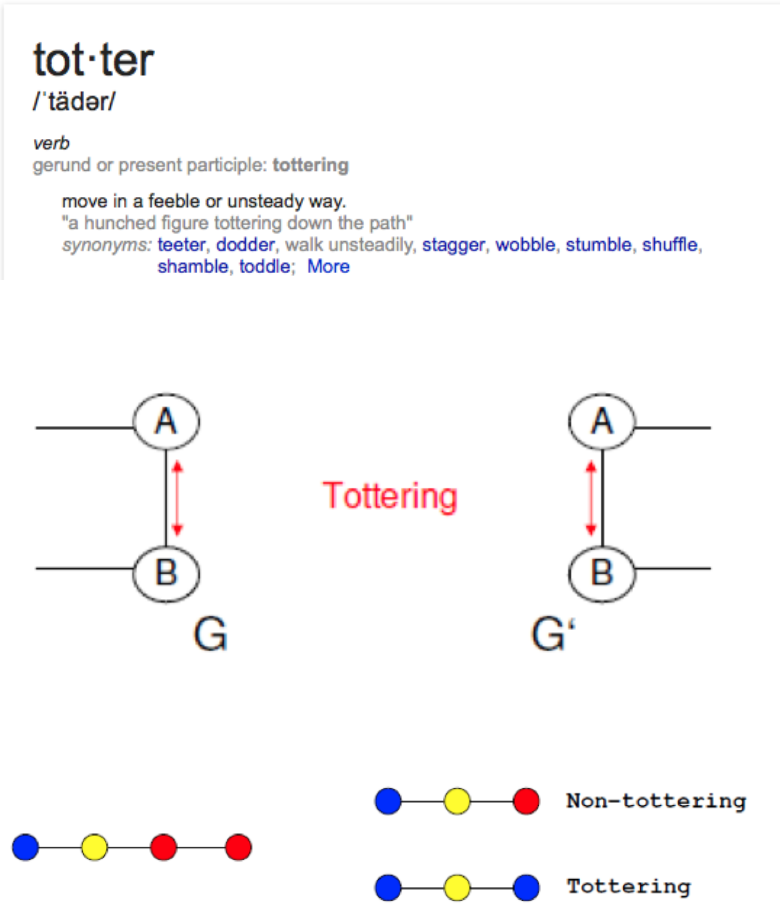
Computing the Random Walk Kernel

Given two graphs G_1 and G_2, with n nodes each

- Computing product graph requires comparison of all pairs of edges in G_1 and $G_2 \implies O(n^4)$ time
- Computing powers of adjacency matrix: matrix multiplication and/or inversion for $n^2 * n^2$ matrix
- runtime $O((n^2)^3) = O(n^6)$!
- Polynomial time, but far from efficient

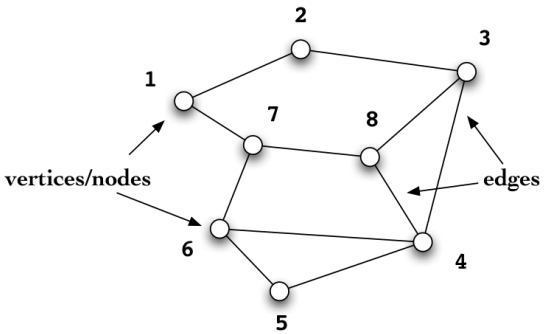
Limitations of walk kernels: Tottering

- Walk kernels allow walks to visit same edges and nodes multiple times
- Each unique walk will generate a new feature
- The feature space will contain a feature for each non-tottering walk plus a large number of tottering walks

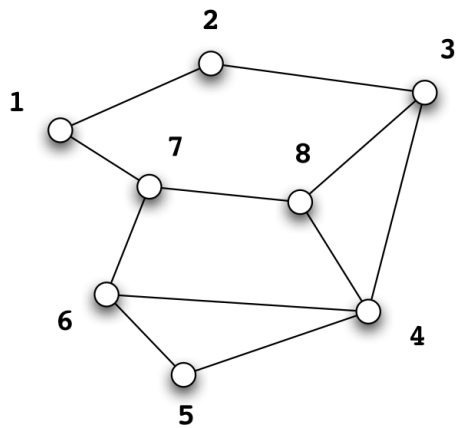


Tottering example

- In our example, all counts in A^2 along the diagonal represents counts of tottering walks $A^2_{ii} = |\{v_i \mapsto v_h \mapsto v_i, h \neq i\}|$



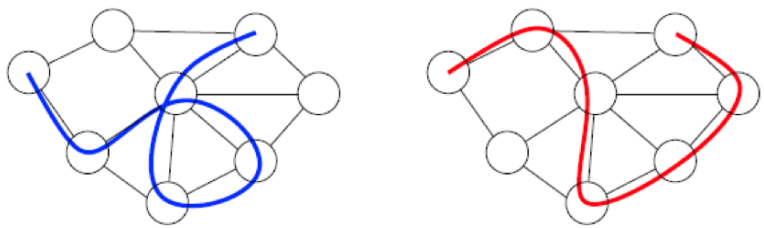
$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$



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

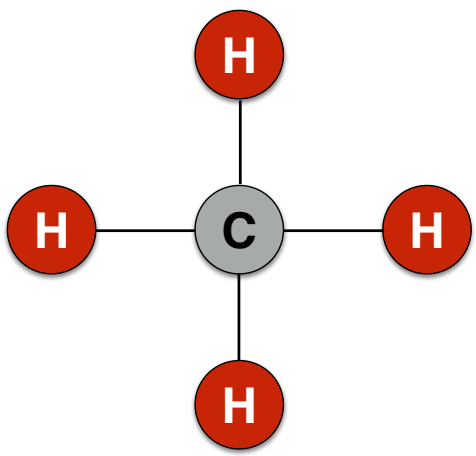
From walks to paths

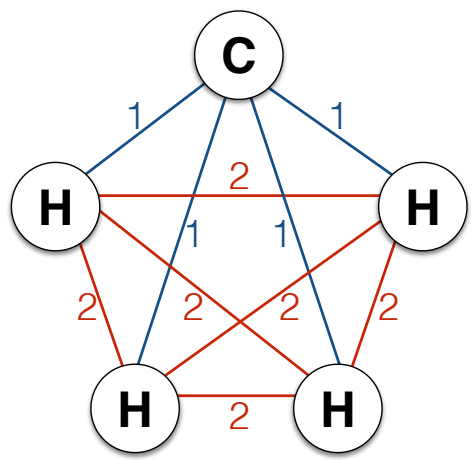
- Remember the basic definitions: A *walk* w of length $k-1$ in a graph is a sequence of nodes $w = (v_1, v_2, \cdots, v_k)$ where $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$.
- w is a *path* if $v_i \neq v_j$ for $i \neq j$.
- Path kernels are based on counting commons paths rather than walks
- Do not suffer from tottering
- But most formulations of path kernels lead to NP-hard algorithms...



Shortest path kernel[2]

- Kernels based on shortest paths are a notable exception: can be computed efficiently
- Shortest paths kernels make use of **Floyd-Warshall transformation** of an input graph G into a shortest-paths graph S in $O(n^3)$ time:
  - S contains the same set of nodes as the graph G
  - There exists an edge between all nodes in S which are connected by a walk in G
  - Every edge in S between two nodes is labeled by the shortest distance between these two nodes.





[2] Borgwardt and Kriegel, 2005

Shortest-path kernel

- Let $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ be the FW-transformations of the graphs G_1 and G_2.
- The shortest-path graph kernel is given

$$\kappa_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} \kappa_{node}(v_1, v_2) \cdot \kappa_{edge}(e_1, e_2) \cdot \kappa_{node}(u_1, u_2)$$

  denoting edges by $e_i = (v_i, u_i) \in E_i$
- Node kernel (κ_{node}) and edge kernels (κ_{edge}) compare the labels of the nodes and edges, respectively:
  - Exact matching: require labels to be the same
  - Soft matching: some labels can be similar to other labels
  - Factored representation: can use feature vectors representing different factors of the nodes/edges
- Kernel computation in $O(|E_1||E_2|) = O(n^4)$ where $n = \max(|V_1|, |V_2|)$

Graphhopper kernel[3]

- Graphhopper kernel uses an alternative formulation on shortest paths:

$$\kappa_{gh}(G_1, G_2) = \sum_{\pi \in P_1} \sum_{\pi' \in P_2} \kappa_p(\pi, \pi')$$

  where P_1 and P_2 are sets of shortest paths between pairs of nodes $\pi = (v_1, \ldots, v_k)$
- Similarity of paths is computed as a sum of node similarities (instead of product), requiring the paths to be of the same length: $\kappa_p(\pi, \pi') = \sum_{j=1}^{k} \kappa_{node}(\pi(j), \pi'(j)) \mathbf{1}_{\{|\pi|=|\pi'|\}}$
- Can be computed in time $O(n^2(m + \log n + \delta^2 + d))$, where m is the number of edges, δ is the diameter of the graph, and d is the dimension of node label vectors
- Currently state of the art performance in several problems

[3] Feragen et al., 2013

Summary

We looked at two different machine learning settings on graphs

- Link prediction concerns the data points as nodes of a large graph
  - Local approach, predicting the neighbouring edges of each node separately
  - Gobal approach, predicting for all pairs of nodes with the same model, with a TPPK kernel
- Graph classification, where a large set of (small) graphs is used to train a classifier using kernels on pairs of graphs
  - Walk kernels
  - Path kernels