Lecture 2: Traditional ML in Graphs

2.1 Traditional Feature-based Methods: Node-level

Review

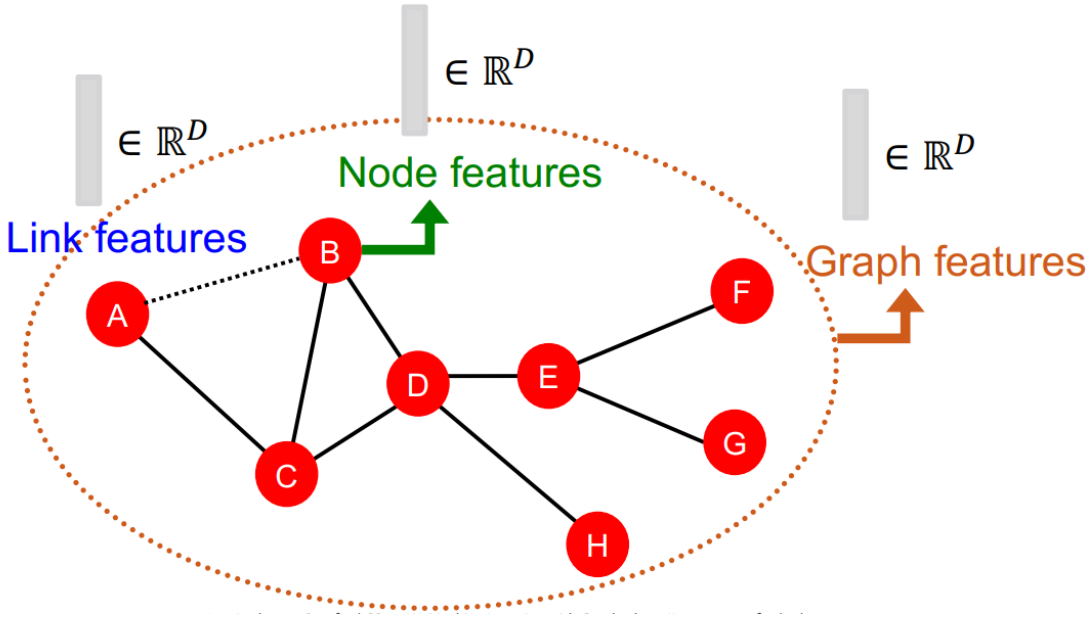
Traditional ML pipeline: design and obtain features for all training data regrading nodes, links and graphs, then train ML model to apply it to make a prediction.

Graph data itself has features, but we are also interested in features that describe its local structure features, that is, the feature describes the topology of the network and structure features can help to make more accurate predictions.

So generally there are two kinds of features:

- Structural features;
- Attributes and properties of nodes or relations.

- Design features for nodes/links/graphs
- Obtain features for all training data



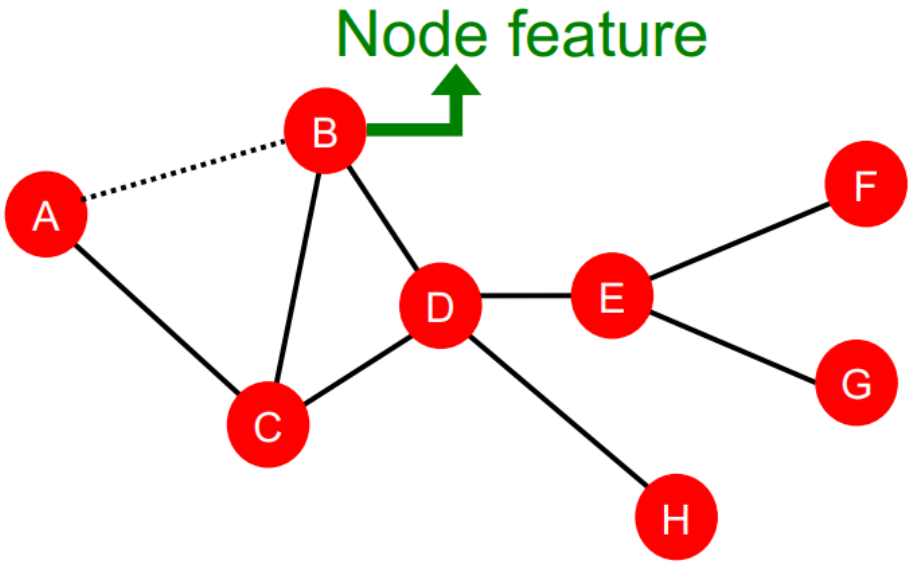
Design Choice

- Features: d-dimensional vectors;
- Objects: Nodes, edges, sets of nodes, entire graphs;
- Objective functions: what tasks are we aiming to solve?

Node-Level Features:

Goal: Characterize the structure and position of a node in the network:

- Node degree
- Node centrality
- Clustering coefficient
- Graphlets



Node Degree

The degree k_v of node v is the number of edges (neighboring nodes) the nodes has.

Node Centrality

- Why: node degree treat all neighboring nodes equally so it cannot capture their importance;
- Node Centrality c_v takes the node importance in a given graph into account;
- Different ways to model importance

1. Eigenvector

- **Eigenvector centrality:**
  - A node v is important if surrounded by important neighboring nodes $u \in N(v)$.
  - We model the centrality of node v as the sum of the centrality of neighboring nodes:

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u$$

λ is some positive constant

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \quad \Longleftrightarrow \quad \lambda \boldsymbol{c} = \boldsymbol{A}\boldsymbol{c}$$

λ is some positive constant

- $\boldsymbol{A}$: Adjacency matrix $\boldsymbol{A}_{uv} = 1$ if $u \in N(v)$
- $\boldsymbol{c}$: Centrality vector

- We see that centrality is the **eigenvector**!

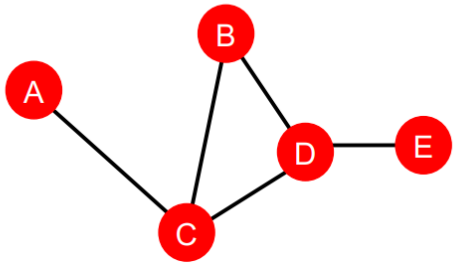
2. Betweenness

Betweenness centrality:

- A node is important if it lies on many shortest paths between other nodes.

$$c_v = \sum_{s \neq v \neq t} \frac{\#(\text{shortest paths betwen } s \text{ and } t \text{ that contain } v)}{\#(\text{shortest paths between } s \text{ and } t)}$$

- Example:



$c_A = c_B = c_E = 0$

$c_C = 3$

(A-C-B, A-C-D, A-C-D-E)

$c_D = 3$

(A-C-D-E, B-D-E, C-D-E)

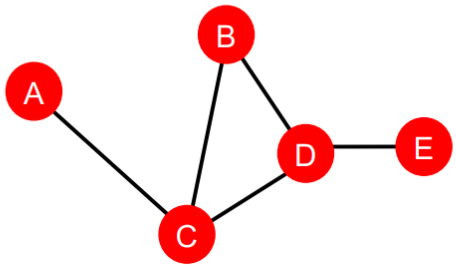
3. Closeness

Closeness centrality:

- A node is important if it has small shortest path lengths to all other nodes.

$$c_v = \frac{1}{\sum_{u \neq v} \text{shortest path length between } u \text{ and } v}$$

- **Example:**



$c_A = 1/(2+1+2+3) = 1/8$

(A-C-B, A-C, A-C-D, A-C-D-E)

$c_D = 1/(2+1+1+1) = 1/5$

(D-C-A, D-B, D-C, D-E)

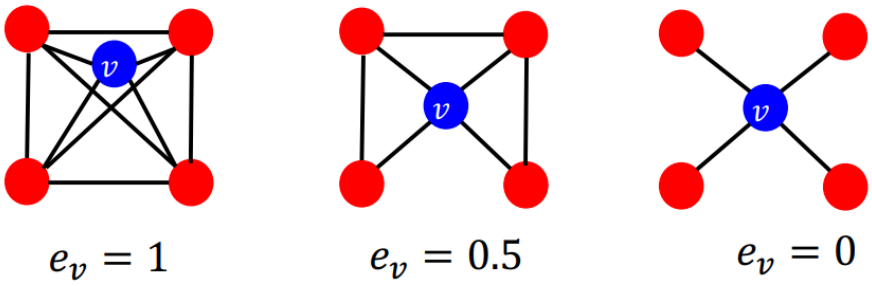
Clustering Coefficient

It measures how connected $v's$ neighboring nodes are:

$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\binom{k_v}{2}} \in [0,1]$$

$\#(\text{node pairs among } k_v \text{ neighboring nodes})$

- **Examples:**

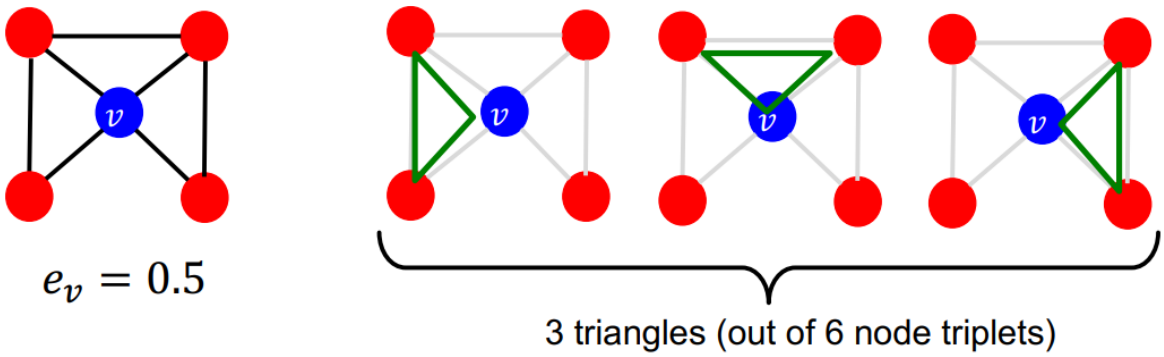


$e_v = 1$ $e_v = 0.5$ $e_v = 0$

The first example: $e_v = \frac{6}{6}$
The second example: $e_v = \frac{3}{6}$
The third example: $e_v = \frac{0}{6}$

Graphlets

Observation: Clustering coefficient counts the #(triangles) in the ego-network



$e_v = 0.5$

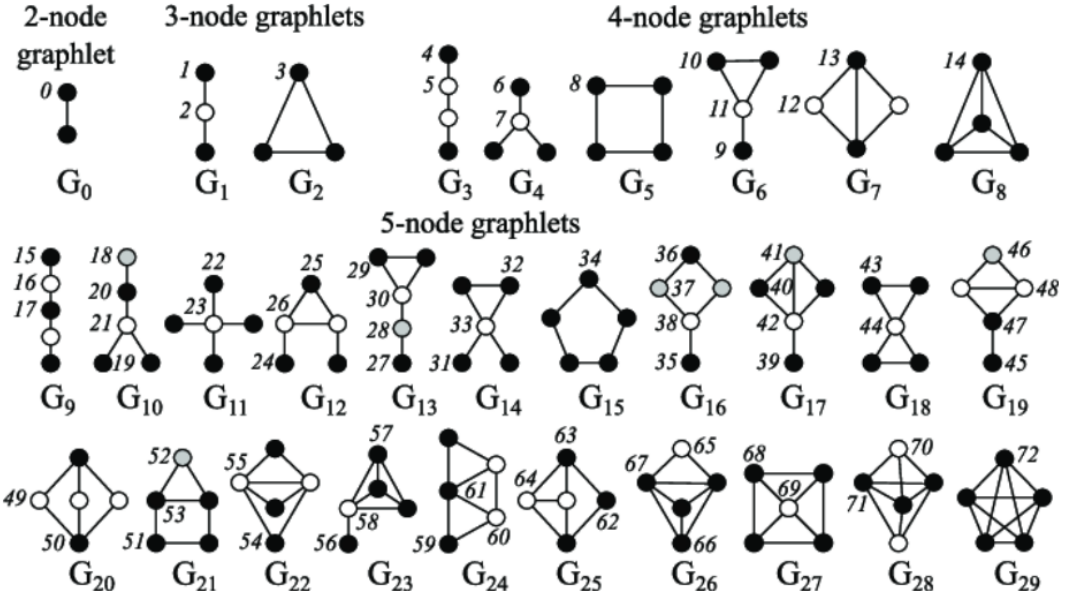
Ego-network of a given node is simply a network that is induced by the node itself and its neighbors. So it's basically degree 1 neighborhood network around a a given node.

We can generalize the above by counting #(pre-specified subgraphs, i.e., **graphlets**)

There are many such triangles in social networks, because it is conceivable that your friends may know each other through your introduction, thus constructing such a triangle/triple.

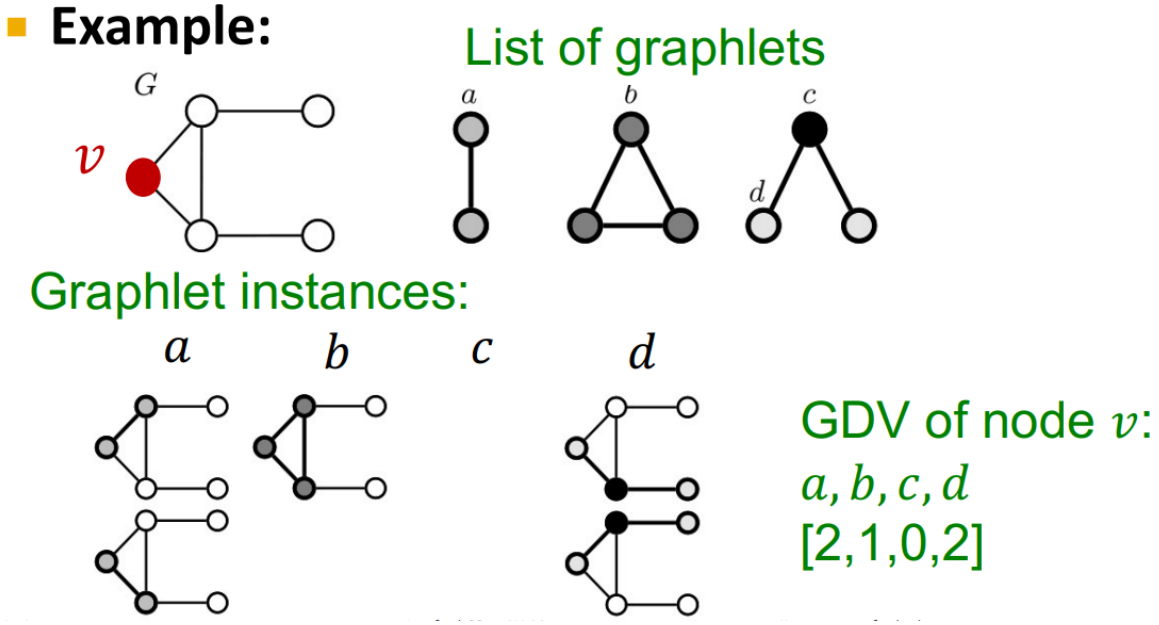
This triangle can be extended to some predefined subgraph pre-specified subgraph, such as the graphlet shown below:

Graphlets: Rooted connected non-isomorphic subgraphs:



- **Graphlet Degree Vector (GDV):** Graphlet-based features for nodes, it counts **graphlets** that a nods touches.;
- **Degree** counts **edges** that a node touches;
- **Clustering coefficient** counts **triangles** that a node touches

- **Example:**



Node-Level Summary

- Importance-based features:
  - Node degree
  - Different node centrality measures

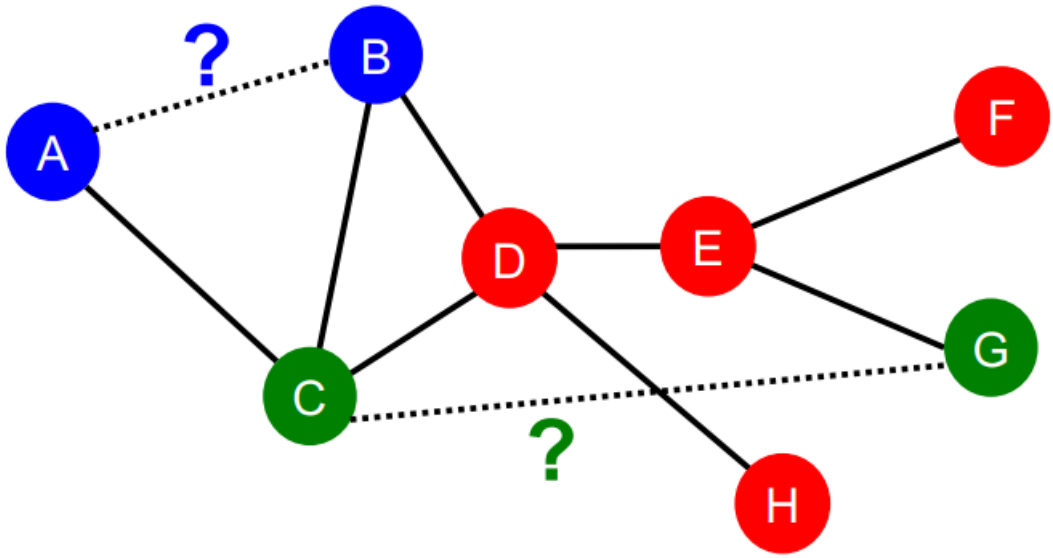
  It can be used to predict celebrity users in a social network.
- Structure-based features:
  - Node degree
  - Clustering coefficient

- Graphlet count vector

It can be used to predict protein functionality in a PPI network.

2.2 Traditional Feature-based Methods: Link-level

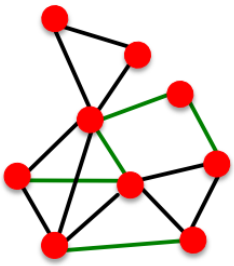
Link prediction tasks can be formulated as predicting new links given existing links. When testing, all node pairs (no existing links) are ranked, and top K node pairs are predicted. Key challenge is to design a pair of nodes.



There are two different tasks:

- Links missing at random
- Links over time

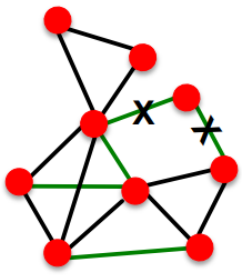
- **1) Links missing at random:**
  - Remove a random set of links and then aim to predict them
- **2) Links over time:**
  - Given $G[t_0, t_0']$ a graph on edges up to time t_0', output a ranked list L of links (not in $G[t_0, t_0']$) that are predicted to appear in $G[t_1, t_1']$
  - **Evaluation:**
    - $n = |E_{new}|$: # new edges that appear during the test period $[t_1, t_1']$
    - Take top n elements of L and count correct edges



$G[t_0, t_0']$
$G[t_1, t_1']$

Link Prediction via Proximity

- **Methodology:**
  - For each pair of nodes *(x,y)* compute score *c(x,y)*
    - For example, *c(x,y)* could be the # of common neighbors of *x* and *y*
  - Sort pairs *(x,y)* by the decreasing score *c(x,y)*
  - **Predict top n pairs as new links**
  - **See which of these links actually appear in $G[t_1, t_1']$**

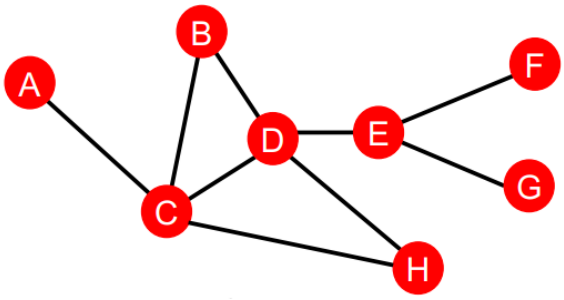


Link-Level Features

1. **Distance-based Feature**

   it measures the shortest-path distance between tow nods.

- Example:



$$S_{BH} = S_{BE} = S_{AB} = 2$$
$$S_{BG} = S_{BF} = 3$$

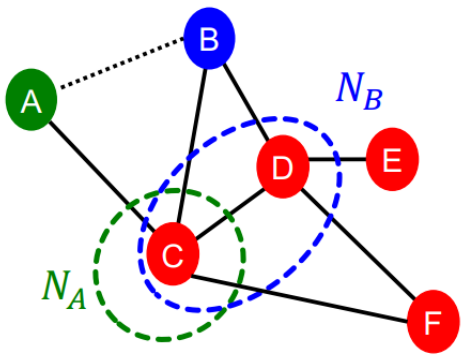
Cons: it does not capture the degree of neighborhood overlap.

2. **Local Neighborhood Overlap**

   It captures neighborhood bodes shared between two nodes $v1$ and $v2$.

- **Common neighbors:** $|N(v_1) \cap N(v_2)|$
  - Example: $|N(A) \cap N(B)| = |\{C\}| = 1$
- **Jaccard's coefficient:** $\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|}$
  - Example: $\frac{|N(A) \cap N(B)|}{|N(A) \cup N(B)|} = \frac{|\{C\}|}{|\{C,D\}|} = \frac{1}{2}$
- **Adamic-Adar index:**

  $\sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)}$
  - Example: $\frac{1}{\log(k_C)} = \frac{1}{\log 4}$



The problem with common neighbors is that point pairs with higher degrees will have higher results, and Jaccard's coefficient is the normalized result.

The Adamic-Adar index performs well in practice. The reason for performing well on social networks: A group of low-degree mutual friends scores higher than a group of celebrity mutual friends.

3. **Global Neighborhood Overlap**

   Limitation of local neighborhood features:

   - Metric is always zero if the two nodes do not have any neighbors in common;
   - the two nodes may still potentially be connected in the future.

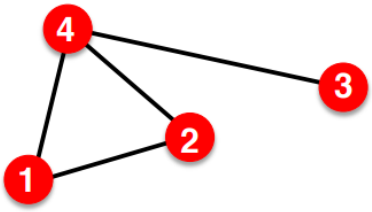
   Global neighborhood overlap metrics resolve the limitation by considering the entire graph.

   Katz index: count the number of paths of all lengths between a given pair of nodes.

   How to use graph adjacency matrix to compute paths between tow nodes:

- **Recall**: $A_{uv} = 1$ if $u \in N(v)$
- Let $P_{uv}^{(K)}$ = #paths of length K between u and v
- We will show $P^{(K)} = A^k$
- $P_{uv}^{(1)}$ = #paths of length 1 (direct neighborhood) between u and $v = A_{uv}$

$P_{12}^{(1)} = A_{12}$



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

- $P_{uv}^{(2)} = \sum_i A_{ui} * P_{iv}^{(1)} = \sum_i A_{ui} * A_{iv} = A_{uv}^2$

Node 1's neighbors

#paths of length 1 between Node 1's neighbors and Node 2

$P_{12}^{(2)} = A_{12}^2$

$$A^2 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 3 \end{pmatrix}$$

Power of adjacency

Discount factor β will assign a smaller weight to long distance path exponentially.

- **Katz index** between v_1 and v_2 is calculated as

Sum over *all path lengths*

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1 v_2}$$

#paths of length l between v_1 and v_2

$0 < \beta < 1$: discount factor

- Katz index matrix is computed in closed-form:

$$\boldsymbol{S} = \sum_{i=1}^{\infty} \beta^i \boldsymbol{A}^i = \underbrace{(\boldsymbol{I} - \beta \boldsymbol{A})^{-1}}_{= \sum_{i=0}^{\infty} \beta^i \boldsymbol{A}^i \text{ by geometric series of matrices}} - \boldsymbol{I},$$

Summary

- Distance-based features:

  Uses the shortest path length between two nodes but does not capture how neighborhood overlaps.

- Local neighborhood overlap:
  - Captures how many neighboring nodes are shared by two nodes.
  - Becomes zero when no neighbor nodes are shared.
- Global neighborhood overlap:
  - Uses global graph structure to score two nodes.
  - Katz index counts #paths of all lengths between two nodes.

2.3 Traditional Feature-based Methods: Graph-level

Goal: We want features that characterize the structure of an entire graph

Kernel Methods

Idea: Design **Kernels instead of feature vectors.**

A quick introduction to Kernels:

- Kernel $K(G,G\prime \in R) $ measures similarity b/w data
- Kernel $\mathbf{K} = (K(G,G\prime))_{G,G\prime} $ must always be positive semidefinite (has positive eigenvals).
- There exists a feature representation $\phi()$ such that $K(G, G\prime) = \phi(G)^T \phi(G\prime)$.
- Once the kernel is defined, off-the-shelf ML model, such as kernel SVM, can be used to make predictions.

$\phi()$ is a representation vector, which may not need to be calculated explicitly.

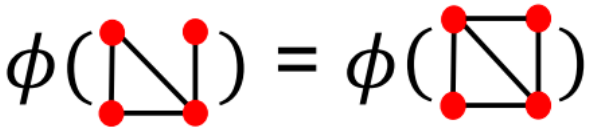
Graph Kernel: Key Idea

Goal: Design graph feature vector $\phi(G)$

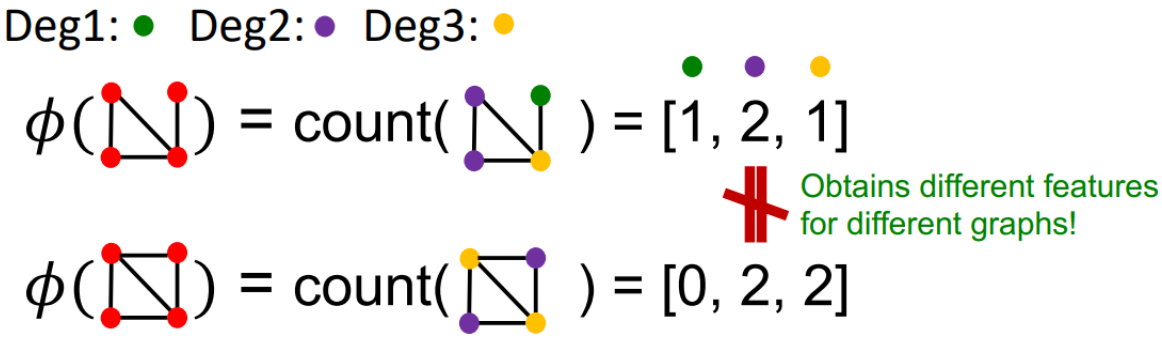
Use **Bag of Words (BoW)** for a graph. BoW simply uses the word counts as features for documents.

Naïve extension to a graph: **Regards nodes as words.**

For following two graphs which have 4 red nodes, we get the same feature vector for these two graphs:



So we could consider using **Bag of node degrees**:



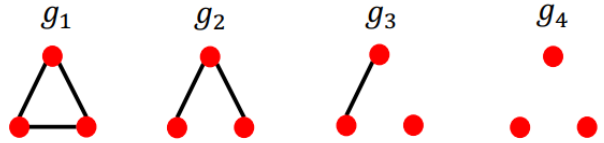
Both **Graphlet Kernel** and **Weisfeiler-Lehman Kernel** use **Bag of *** representation of graph, where ***** is more sophisticated than node degrees.

Graphlet Features

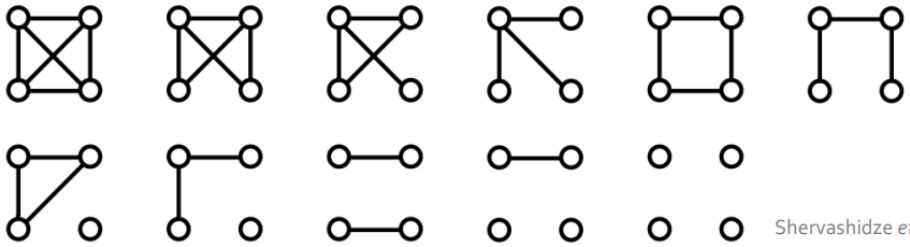
- **Key idea:** Count the number of different graphlets in a graph
- **Difference from node-level features:**
  - Nodes in graphlets here do not need to be connected (allows for isolated nodes);
  - The graphlets here are not rooted.

Let $G_k = (g_1, g_2, \ldots, g_{n_k})$ be a list of graphlets of size k.

- For $k = 3$, there are 4 graphlets.



- For $k = 4$, there are 11 graphlets.

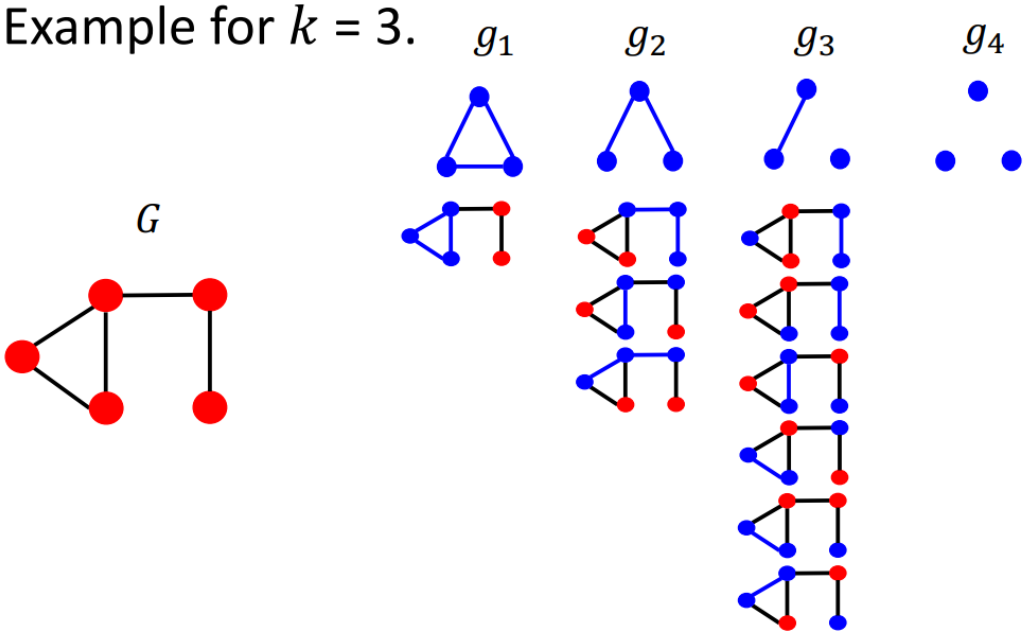


Shervashidze *et al.*, AISTATS 2011

- **Graphlet count vector:** Given graph G, and a graphlet list $G_k = (g_1, g_2, \ldots, g_{n_k})$, define the graphlet count vector as:

$$(\boldsymbol{f}_G)_i = \#(g_i \subseteq G) \text{ for } i = 1,2,\ldots,n_k.$$

- **Example:**
  - Example for k = 3.



$$\boldsymbol{f}_G = (1, \quad 3, \quad 6, \quad 0)^{\mathrm{T}}$$

- For different sizes that will greatly skew the value, we normalize each feature vactor:

$$\boldsymbol{h}_G = \frac{\boldsymbol{f}_G}{\mathrm{Sum}(\boldsymbol{f}_G)} \qquad K(G, G') = \boldsymbol{h}_G{}^{\mathrm{T}} \boldsymbol{h}_{G'}$$

- **Limitations:** Counting graphlets is expensive.

Weisfeiler-Lehman Kernel

Goal: design an efficient graph feature descriptor $\phi(G)$.

Idea: use neighborhood structure to iteratively enrich node vocabulary.

Color Refinement

- Assign an initial color to each node v and then iteratively refine node colors by

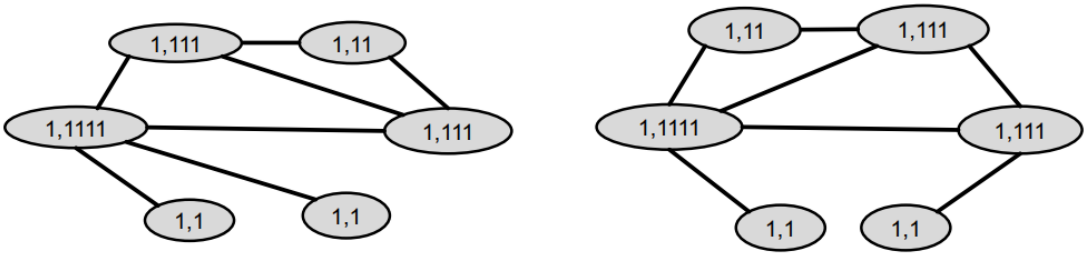
$$c^{(k+1)}(v) = \text{HASH}\left(\left\{c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right\}\right),$$

where HASH maps different inputs to different colors.

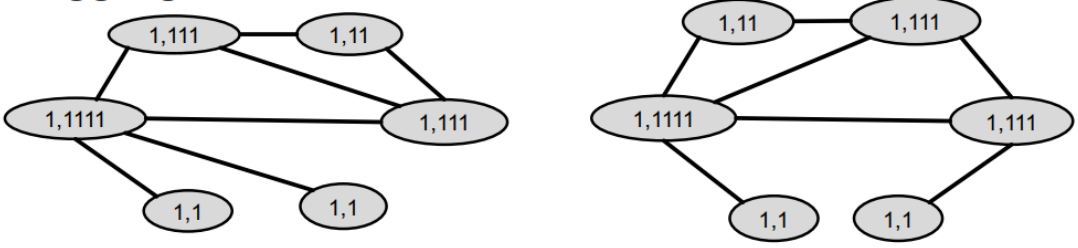
After K steps of refinement, $c^{K}(v)$ summarizes the structure of K-hop neighborhood.

- **Example**
  - Assign initial colors

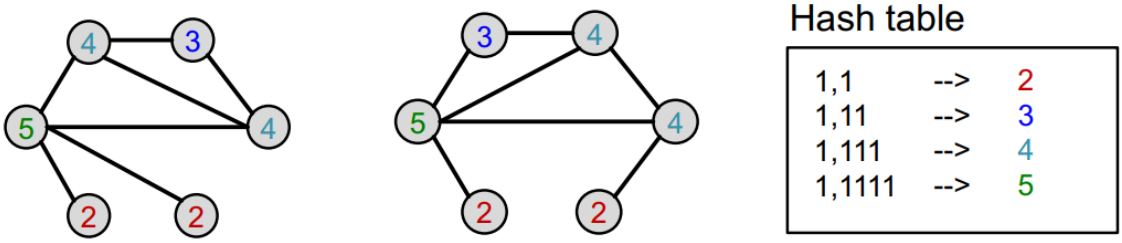
  - Aggregate neighboring colors



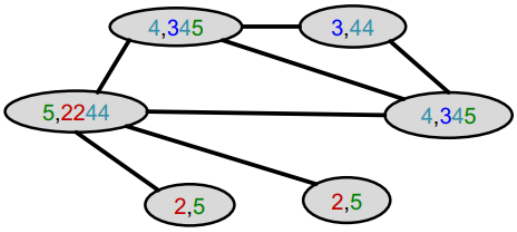
  - Aggregated colors

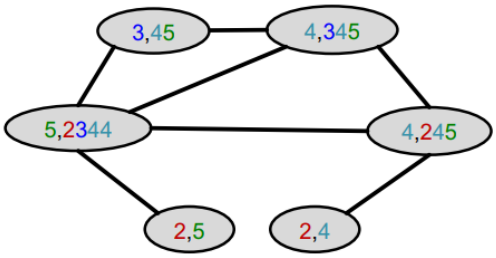


  - Hash aggregated colors

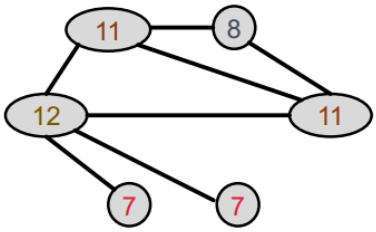


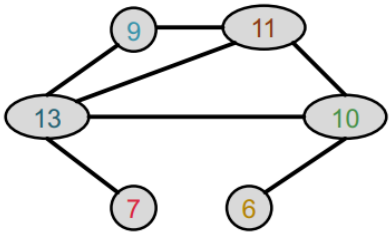
- Aggregated colors





- Hash aggregated colors

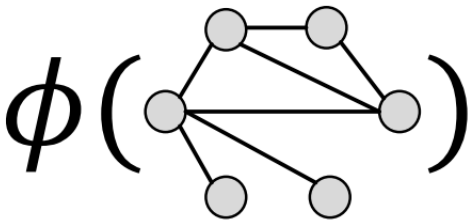




Hash table

2,4	-->	6
2,5	-->	7
3,44	-->	8
3,45	-->	9
4,245	-->	10
4,345	-->	11
5,2244	-->	12
5,2344	-->	13

After color refinement, WL kernel counts number of nodes with a given color.

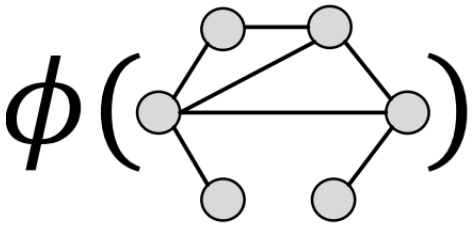


Colors

1,2,3,4,5,6,7,8,9,10,11,12,13

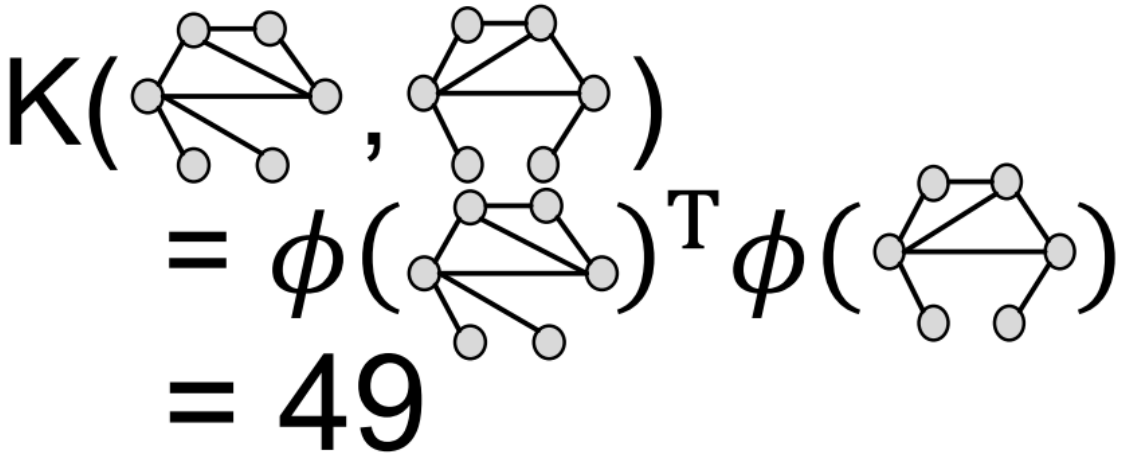
= [6,2,1,2,1,0,2,1,0, 0, 0, 2, 1]

Counts



1,2,3,4,5,6,7,8,9,10,11,12,13

= [6,2,1,2,1,1,1,0,1, 1, 1, 0, 1]



- **Pros**: Computationally efficient