Graph Machine Learning

Lecture 7 : Shallow Graph Learning

Sept 9 2026

Xavier Bresson

https://x.com/xbresson

Department of Computer Science

National University of Singapore (NUS)

Course lectures

- Introduction to Graph Machine Learning
- Part 1: GML without feature learning (before 2014)
  - Introduction to Graph Science
  - Graph Analysis Techniques without Feature Learning
    - Graph clustering
    - Graph SVM
    - Recommendation on graphs
    - Graph-based visualization
- Part 2 : GML with shallow feature learning (2014-2016)
  - Shallow graph learning
- Part 3 : GML with deep feature learning, a.k.a. GNNs (after 2016)
  - Graph Convolutional Networks (spectral and spatial)
  - Weisfeiler-Lehman GNNs
  - Graph Transformer & Graph ViT
  - Graph generation & molecular science
  - Material design
  - Integrating GNNs and LLMs
  - Deep Learning for Combinatorial optimization

Outline

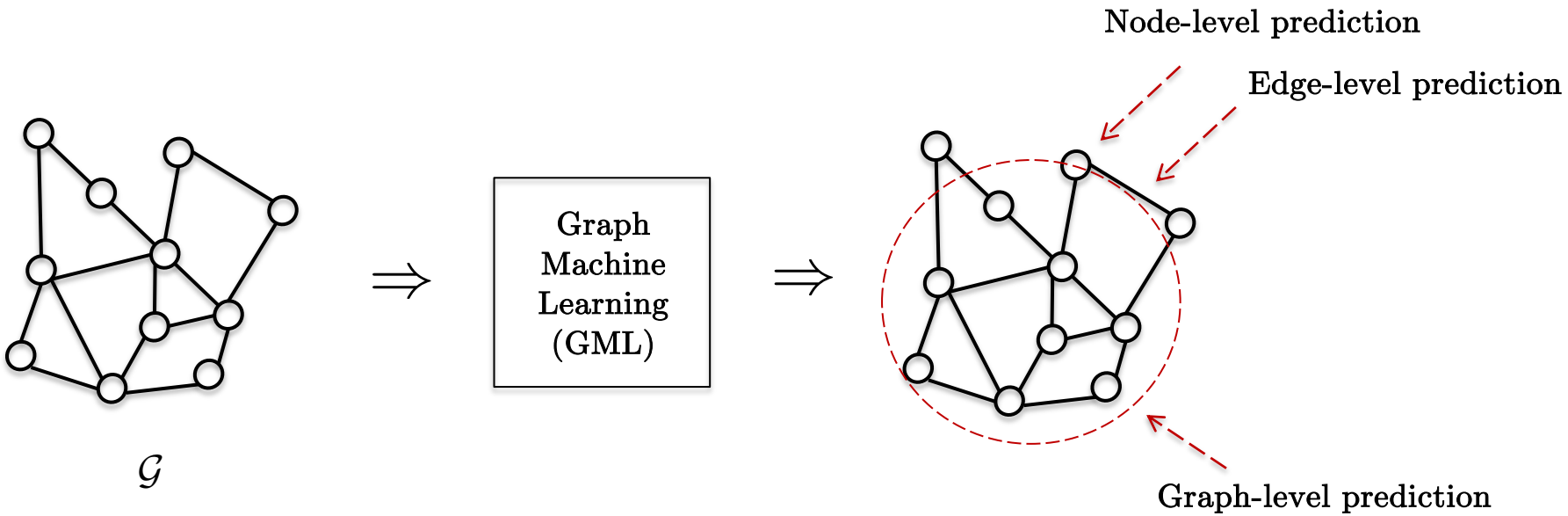
- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

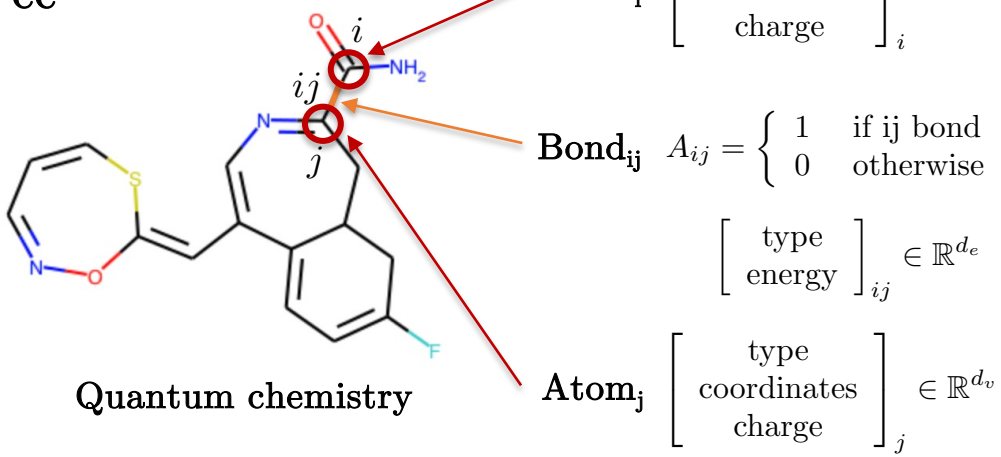
Prediction tasks on graphs

- There are three fundamental levels of prediction on graphs :
  - Node-level prediction, e.g. identifying a fraudulent account in a financial network.
  - Edge/link-level prediction, e.g. detecting a fraudulent transaction.
  - Graph/subgraph-level prediction, e.g. identifying a money-laundering ring.



Designing graph features

- A key requirement for successful node, edge and graph-level prediction tasks
  - Expressive, transferable / generalizable representation features
- Type of raw input features
  - Intrinsic/structural features : node features (node degree), link features (pairwise distance between nodes), graph features (graph motifs).
  - Extrinsic data features : node features (atom type, 3D coordinates), edge features (bond type), graph features (molecular charge, energy).
- Evolution of graph representation learning
  - Before 2014 : Engineered/hand-crafted features, e.g. PageRank
  - 2014-2016 : Shallow learned features, e.g. Node2Vec
  - After 2016 : Deep learned features, e.g. GNNs

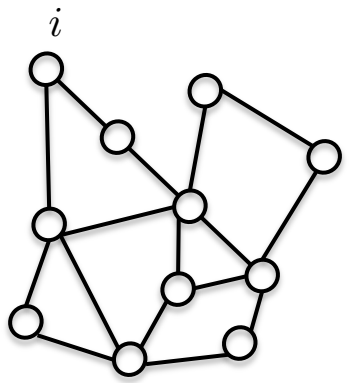


Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Node-level features

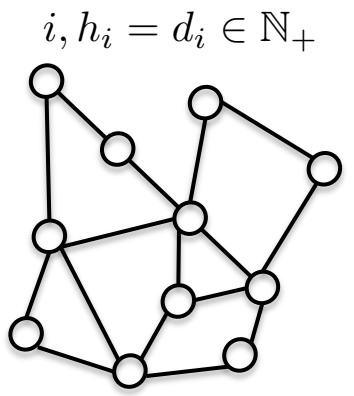
- There are two complementary sources of information to characterize nodes in a graph.
  - Structural information : Topology-derived properties, e.g. node degree or centrality.
  - Positional information : Graph-relative or geometric position, e.g. positional encodings or 3D atomic coordinates.



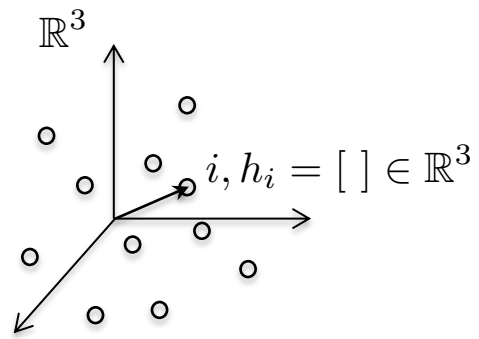
$\mathcal{G}$

Node feature

$\Rightarrow$



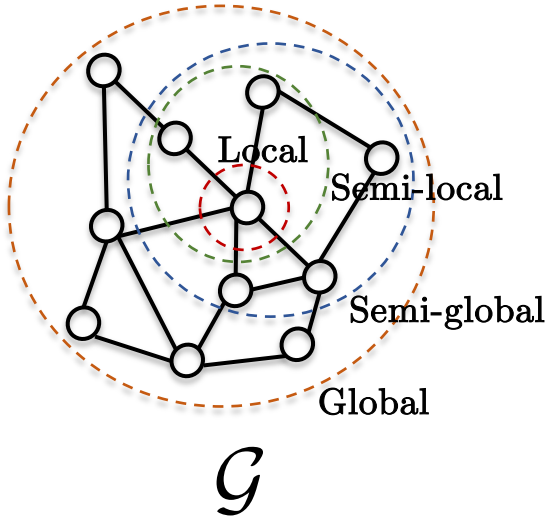
Structural information
e.g. node degree



Positional information
e.g. Euclidean coordinates

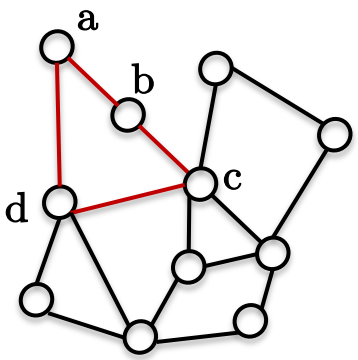
Structural node features

- Structural node features are derived from the topology of the graph.
  - Local properties of graphs
    - E.g. node degree : The number of neighboring nodes directly connected to a node.
  - Semi-local/neighborhood-level properties
    - E.g. graphlet degree vector : Characterizes the topology of a node's local neighborhood. (discussed later)
  - Global path-based properties
    - E.g. betweenness centrality : Measures how often a node lies on shortest paths between other nodes.
  - Global spectral/diffusion properties
    - E.g. eigenvector centrality s.a PageRank : Measures a node's influence through the connectivity of the entire graph. (discussed previously)



Graphlets

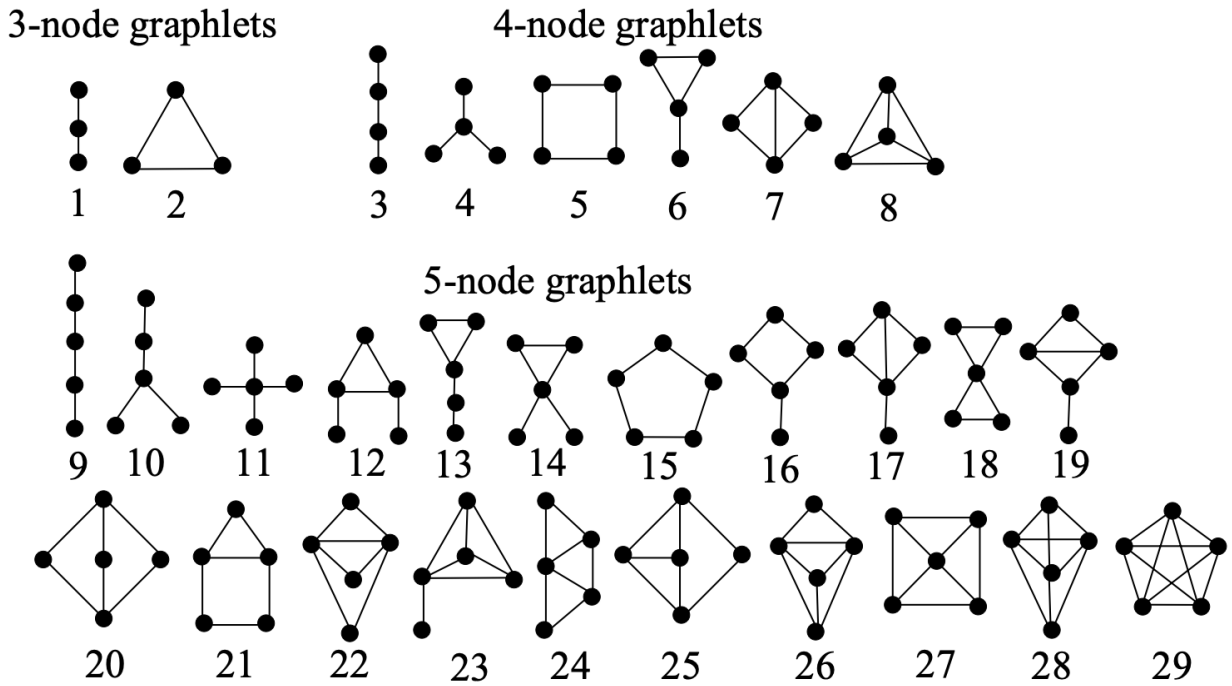
- Graphlets[1] : Small, connected and induced subgraphs that capture local graph structure.
- An induced subgraph is formed by selecting a set of nodes and keeping all edges between them that exist in the original graph.
- For example, the induced subgraph of nodes {a,b,c,d} is a cycle.
- A graph and its nodes can be characterized by the types and frequencies of graphlets it contains.
- The number of graphlets grows as $O(2^{\binom{k}{2}}) = O(2^{k^2})$, i.e. super-exponentially with graphlet size k.
- Brute-force counting of all k-node graphlets has $O(\binom{N}{k}) = O(N^k)$ complexity, i.e. polynomial in N for fixed k, but exponential in the graphlet size k.



[1] Przulj, Corneil, Jurisica, Modeling Interactome, Scale-Free or Geometric, 2004

Graphlets

- All 3-node, 4-node, and 5-node graphlets, ordered from the least to the most dense w.r.t. the number of edges[1].

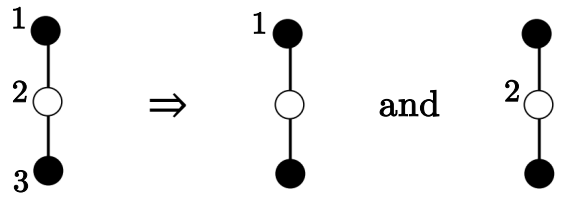


[1] Przulj, Corneil, Jurisica, Modeling Interactome, Scale-Free or Geometric, 2004

Graphlet orbit

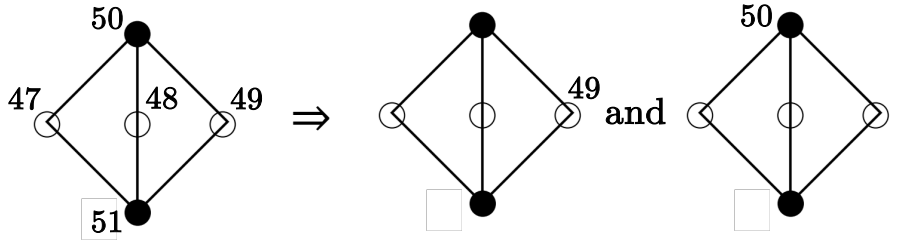
- Graphlet orbit : A set of structurally equivalent nodes within a graphlet, i.e. nodes that can be mapped to one another under graph automorphism.
- Graph automorphism : It is a relabeling of the nodes that leaves the graph unchanged, i.e. adjacency relationships are preserved.
- Two nodes belong to the same orbit if some such relabeling can swap one node with the other.
- For example, nodes 1&3, 50&51 and 47-49 can be changed without changing the graph structure.

Nodes 1&3 belong to the same orbit, and Node 2 belong to another orbit



2-node graphlet G_1 | Orbit 1 of graphlet G_1 | Orbit 2 of graphlet G_1

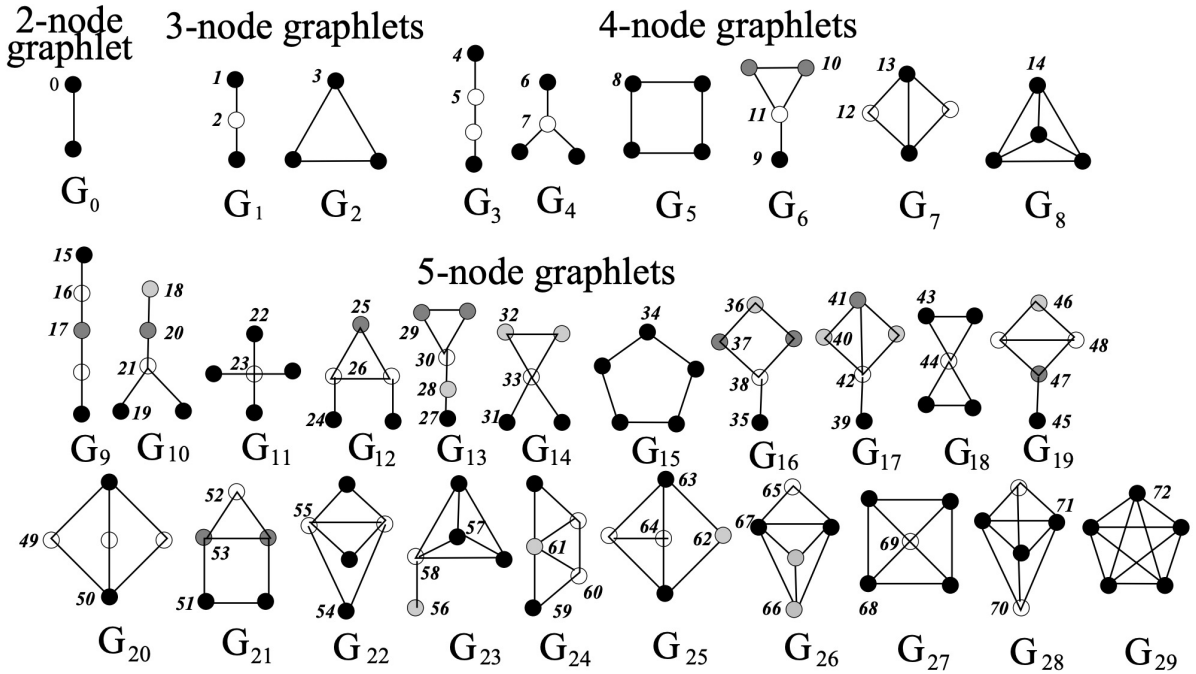
Nodes 50&51 belong to the same orbit, and Nodes 47,48,49 belong to another orbit



5-node Graphlet G_{20} | Orbit 49 of graphlet G_{20} | Orbit 50 of graphlet G_{20}

Graphlet orbits

- The 30 connected 2–5-node graphlets G_i, $i \in \{0, 1, \ldots, 29\}$, contain 73 distinct node orbits, indexed by 0,...,72[1].
- Each node orbit index {0,..,72} identifies a specific structural position within a graphlet.



[1] Milenkovic, Przulj, Uncovering Biological Network Function via Graphlet Degree Signatures, 2008

Graphlet degree vector

- Graphlet Degree Vector (GDV) :
  - A vector h_i counting how many times a node i participates in each graphlet orbit.
  - $h_{i,r}$ counts the number of occurrences in which node i occupies orbit r.
  - GDV provides a detailed multiscale description of the local topology around node i.

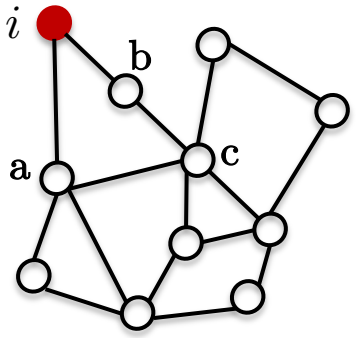
$$\mathbf{h}_i = [h_{i,0}, h_{i,1}, \ldots, h_{i,72}]^T \in \mathbb{N}^{73}$$

with $h_{i,r} = \#\{\text{induced graphlet occurrences containing node } i \text{ in orbit } r\}$

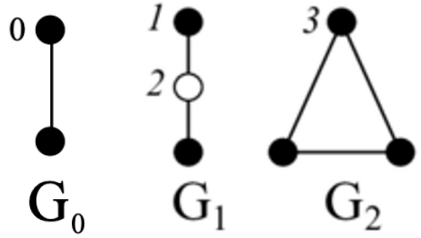
and $r \in \{0, \ldots, 72\},\ 2 \leq k \leq 5$

Example

- Recall that graphlets are induced subgraphs, i.e. once a set of nodes is selected, all edges between those nodes present in the original graph are included.
- Consequence : If the orbit is defined as a node-path, it cannot simultaneously be a node-cycle.
- For example, for orbit G_3^4, the sequence i-a-c-b is not counted because edge b-i closes the path into a cycle. Similarly, for orbit G_3^5 and the sequence a-i-b-c.

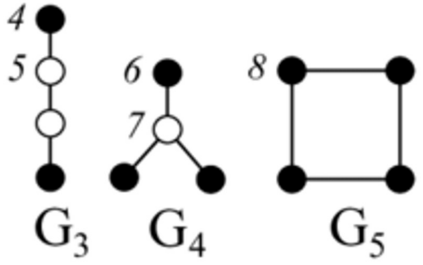


$\mathcal{G}$



2-node graphlet

3-node graphlets



4-node graphlets

$$h_i = \begin{bmatrix} 2 & (\#G_0^0) \\ 4 & (\#G_1^1) \\ 1 & (\#G_1^2) \\ 0 & (\#G_2^3) \\ 8 & (\#G_3^4) \\ 2 & (\#G_3^5) \\ \dots & \end{bmatrix} \in \mathbb{N}^{73}$$

Orbit index

Graphlet index

Graphlet degree vector (GDV)

Lab 1 : Clustering with GDV

- Implement node classification with graphlet degree vector.

Lecture : Shallow Graph Learning

Lab 01 : Node Classification on SBM Graphs with Graphlet Degree Vectors -- Solution

Xavier Bresson

The main steps of this notebook are:

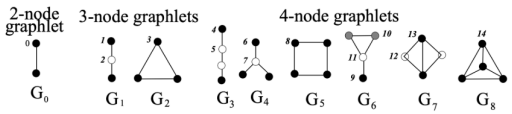
1. Generate a dataset of SBM graphs with **5 clusters** and graph sizes sampled uniformly from **[140, 160]**.
2. Compute a node-level **Graphlet Degree Vector (GDV)** for each node.
3. Train a **linear SVM** supervised node classifier using the GDVs as features.
4. Evaluate the classifier on **new, unseen 5-cluster SBM graphs**.

The training, validation, and test sets are split **at the graph level rather than the node level**.
This means that every node in the test set comes from a graph that the model has **never seen during training**.

Graphlet Degree Vector

We use the **15 automorphism orbits** associated with connected **graphlets on 2–4 nodes**:

- Orbit 0: edge
- Orbits 1–3: roles in the 3-node path and triangle
- Orbits 4–14: roles in the connected 4-node graphlets



For a node v, the GDV is defined as

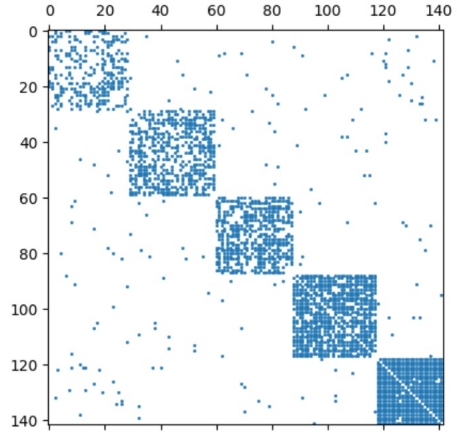
$$\mathbf{h}_v = [h_{v,0}, h_{v,1}, \ldots, h_{v,14}]^\top \in \mathbb{N}^{15},$$

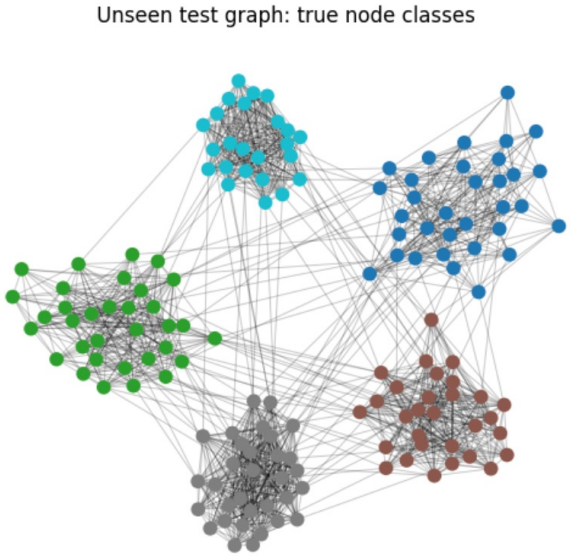
where $h_{v,r}$ counts how many times node v participates in the graphlet orbit r.

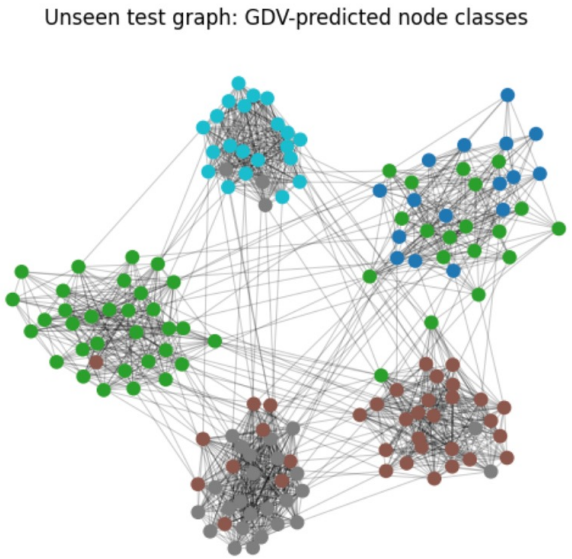
The first four coordinates are computed exactly. But the 4-node orbit counts are estimated by rooted Monte Carlo sampling to keep the notebook fast.

Because graph sizes vary, we also use a graph-size-normalized GDV for classification.

```
Number of nodes: 142
Block sizes: [29, 31, 28, 30, 24]
SBM probability matrix: [[0.35 0.01 0.01 0.01 0.01]
 [0.01 0.5  0.01 0.01 0.01]
 [0.01 0.01 0.65 0.01 0.01]
 [0.01 0.01 0.01 0.8  0.01]
 [0.01 0.01 0.01 0.01 0.95]]
Node labels: [0 0 0 0 0 0 0 0 0 0]
```







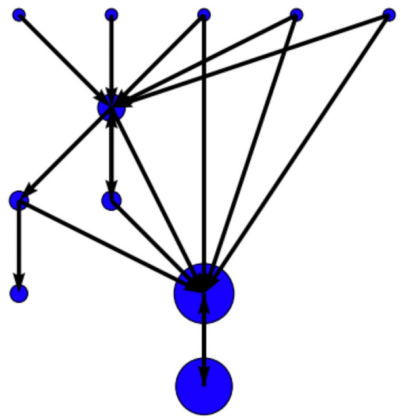
Eigenvector centrality

- A global structural property associated to each node.
  - Measures a node's influence within the network.
  - A node has high centrality when it is connected to other high-centrality nodes.
  - Several spectral centrality measures are based on eigenvectors of graph-related matrices.
  - PageRank[1] is a popular spectral centrality measure based on the dominant eigenvector of a stochastic transition matrix.
  - By the Perron-Frobenius Theorem[2,3], under connectivity conditions, eigenvector centrality is given by a nonnegative dominant eigenvector of the adjacency matrix.

$$Ah = h \in \mathbb{R}^N, \; h_i \in \mathbb{R}_+$$

Adjacency matrix of the graph

Largest eigenvector/ eigenvector centrality

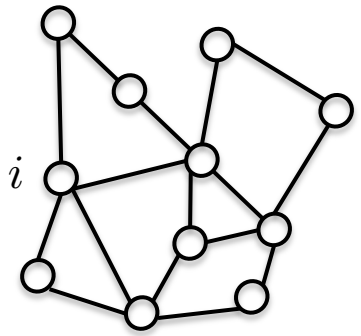


[1] Page, Brin, Motwani, Winograd, The PageRank citation ranking: Bringing order to the web, 1999
[2] Perron, Zur Theorie der Matrices, 1907
[3] Frobenius, Ueber Matrizen aus nicht negativen Elementen, 1912

Eigenvector embedding

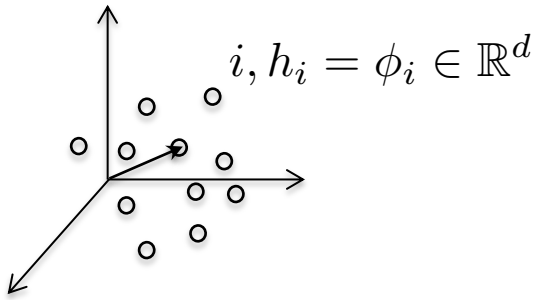
- Geometric features : Raw spatial information, e.g. 3D coordinates or relative orientations of atoms.
- Structural positional features : Node positions derived directly from the structure of the graph, rather than from physical coordinates.
- Spectral embedding : Laplacian eigenvectors[1] map nodes to a d-dimensional Euclidean space while approximately preserving local relationships in the graph.

$$L = \mathrm{I} - D^{-1/2} A D^{-1/2} = \Phi \Lambda \Phi^T \in \mathbb{R}^{N \times N}$$

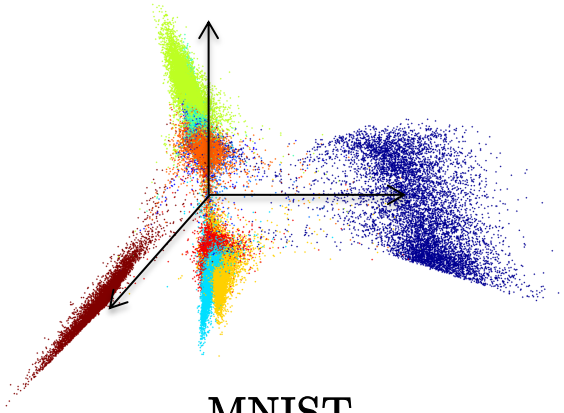


$\mathcal{G}$

$\Rightarrow$



Euclidean embedding of the graph
with Laplacian eigenvectors



MNIST
embedded $\in \mathrm{R}^3$

[1] Belkin, Niyogi, Laplacian eigenmaps for dimensionality reduction and data representation, 2003

Spectral graph clustering

- Compute k clusters as follows :
  - Compute the Normalized Graph Laplacian : $L = I - D^{-1/2} A D^{-1/2} \in R^{N \times N}$, where A is the adjacency matrix
  - Compute its k smallest eigenpairs : $Lu_q = \lambda_q u_q$, $u_q \in R^N$, $1 \leq q \leq k$
  - Form the spectral embedding : $Y^* = (u_1,..,u_k) \in R^{N \times k}$
  - Binarize spectral solution Y^* to obtain discrete clusters :
    - Generally, solution Y^* is not binary, i.e. no cluster can be directly identified.
    - Consider Y^* as k-dim embedding coordinates of X and
    - Apply standard k-means to the rows of Y^* to identify k clusters.
- This procedure is known as spectral clustering[1], where NormalizedCut[2,3] is the most used.

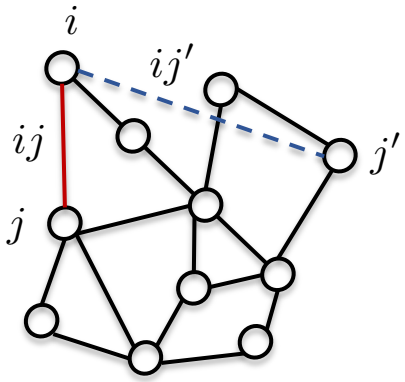
[1] Von Luxburg, A tutorial on spectral clustering, 2007
[2] Shi, Malik, Normalized cuts and image segmentation, 2000
[3] Yu, Shi, Multiclass spectral clustering, 2003

Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Edge-level features

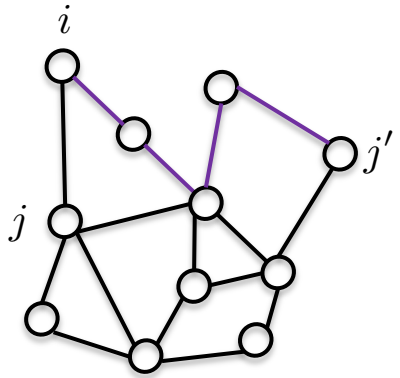
- An edge/link connects a pair of nodes, and its features describe the relationship between them.



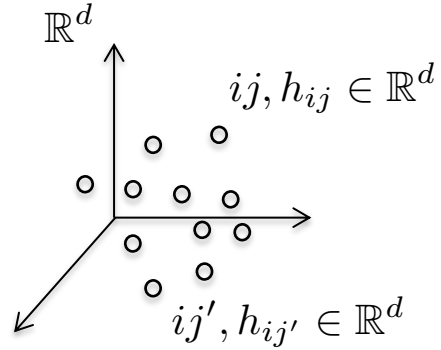
$\mathcal{G}$

Edge features

$\Rightarrow$



Structural information e.g. common neighbors or shortest-path distance for a link (i,j) or link (i,j')

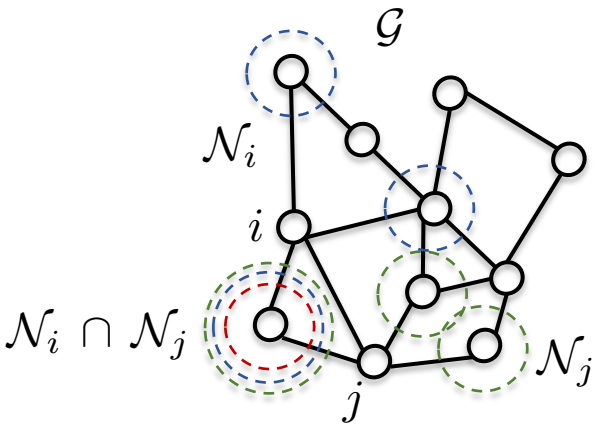


Geometric/positional information e.g. relative position $h_{ij} = h_j - h_i \in R^d$ or scalar distance $||h_j - h_i||$.

Edge-level features

- Edge features derived from graph topology :
  - Local neighborhood properties
    - Jaccard index[1] : Measures the similarity between two nodes as the fraction of neighbors shared by their combined neighborhoods.

$$h_{ij} = \frac{|\mathcal{N}_i \cap \mathcal{N}_j|}{|\mathcal{N}_i \cup \mathcal{N}_j|} \in [0, 1]$$



$$h_{ij} = \frac{1}{5}$$

  - Path- and diffusion-based properties (next slides)
    - Katz index
    - Shortest path
    - Diffusion distance

[1] Jaccard, The Distribution of the Flora in the Alpine Zone, 1912

Katz index

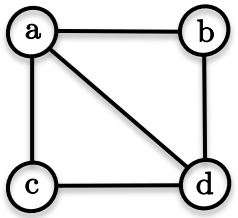
- The Katz index[1] measures the similarity between two nodes by counting walks of all lengths, with longer walks receiving smaller weights.
- The number of walks of length k from node i to node j is given by the k-th power of the adjacency matrix A^k, where each entry $A_{ij} \in \{0,1\}$ is the adjacency matrix.

$$(A^k)_{ij} = \#\{\text{walks of length } k \text{ from } i \text{ to } j\} \in \mathbb{N}$$

$$h_{ij}^{\text{Katz}} = \sum_{k=1}^{\infty} \beta^k (A^k)_{ij}, \; 0 < \beta < \frac{1}{\rho(A)} < 1,$$

where $\rho(A)$ is the spectral radius of A, i.e. $\rho(A) = \max_{\lambda \in \text{eigenvalues}(A)} |\lambda|$.

$\mathcal{G}$



$$A = A^1 = \begin{array}{c} \begin{matrix} \text{a} & \text{b} & \text{c} & \text{d} \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} \text{(node index)} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \end{matrix}$$

$$A^2 = A^1.A^1 = \begin{array}{c} \begin{matrix} \text{a} & \text{b} & \text{c} & \text{d} \end{matrix} \\ \begin{bmatrix} 3 & 1 & 1 & 2 \\ 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 \\ 2 & 1 & 1 & 3 \end{bmatrix} \end{array} \begin{matrix} \\ \text{a} \\ \text{b} \\ \text{c} \\ \text{d} \end{matrix}$$

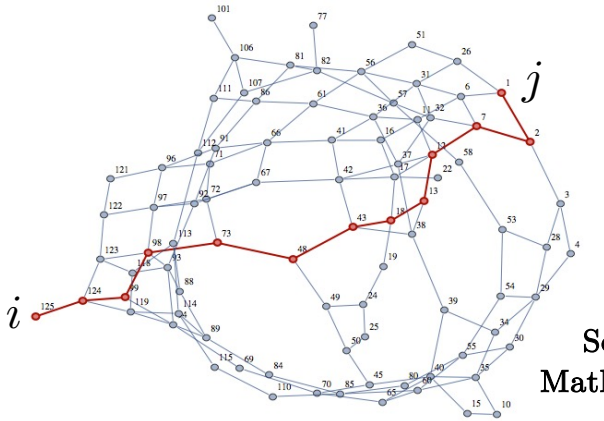
[1] Katz, A New Status Index Derived from Sociometric Analysis, 1953

Shortest path

- The shortest-path distance between two nodes provides a structural feature describing their proximity in the graph.

- Dijkstra's algorithm computes shortest-path distances in $O((E+V)\log V)$ time using a binary-heap priority queue:

$$
\begin{aligned}
&d(s) = 0,\ d(v) = \infty\ (v \neq s),\\
&u \leftarrow \arg\min_{v \notin S} d(v),\\
&d(v) \leftarrow \min\{d(v),\ d(u) + w(u,v)\},\\
&S \leftarrow S \cup \{u\},\\
&d_{ij}^{\text{shortest}} = \min_{p:i\to j} \sum_{e\in p} w_e.
\end{aligned}
$$



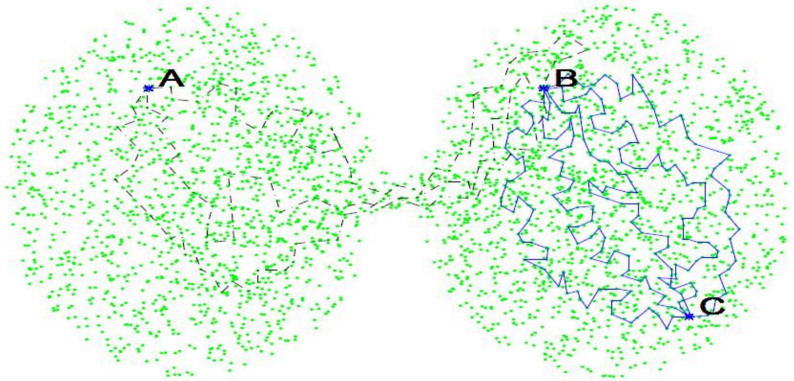
Source: Mathematica

- Using a Fibonacci heap min-priority queue[2], the complexity improves to $O(E + V\log V) = O(E)$.

[1] Dijkstra, A note on two problems in connexion with graphs, 1959
[2] Fredman, Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, 1987

Diffusion distance

- Diffusion distance measures the structural proximity between two nodes through the way information diffuses over the graph.
  - Random walk distance[1] : Compares the probability distributions reached from two nodes after t steps.
  - Heat/Laplacian diffusion distance[2] : Compares the diffusion profiles generated by the graph heat kernel.
- Diffusion distances provide informative multiscale features for link prediction.



Diffusion distance[3]

$$d_{AB}^{\text{Diffusion}} \gg d_{BC}^{\text{Diffusion}}$$

[1] Coifman, Lafon, Diffusion Maps, Applied and Computational Harmonic Analysis, 2006
[2] Kondor, Lafferty, Diffusion Kernels on Graphs and Other Discrete Input Spaces, 2002
[3] Lafon, Diffusion Maps and Geometric Harmonics, 2004

Random walk distance

- Define the random-walk transition matrix : $P = D^{-1}A \in R^{N \times N}$, where P_{ij} is the probability of moving from node i to node j.
- After t steps, $(P^t)_{ik} = Pr(X_t = k \mid X_0 = i)$ gives the probability of reaching node k from node i.
- The diffusion distance[1] between nodes i and j compares their t-step probability distributions :

$$d_t^{\text{diff}}(i,j)^2 = \sum_{k \in V} \left[(P^t)_{ik} - (P^t)_{jk}\right]^2 / \pi_k,$$

where π is the stationary distribution defined as

$$\pi_k = \frac{d_k}{\sum_m d_m}, \qquad P^T \pi = \pi, \qquad \sum_k \pi_k = 1$$

For undirected graphs,

$$d_t^{\text{diff}}(i,j)^2 = \sum_{\ell=1}^{n} \lambda_\ell^{2t} \left[\psi_\ell(i) - \psi_\ell(j)\right]^2, \text{ with } P\psi_\ell = \lambda_\ell \psi_\ell, \ (\lambda_\ell, \psi_\ell) \text{ eigenpairs}$$

- Interpretation : Two nodes are close when random walks starting from them spread similarly over the graph after t steps.

[1] Coifman, Lafon, Diffusion Maps, 2006

Heat/Laplacian diffusion distance

- Define the normalized graph Laplacian : $L = I - D^{-1/2} A D^{-1/2} \in R^{N \times N}$
- The heat kernel[1] $H_t = e^{-tL} \in R^{N \times N}$, $t > 0$, describes continuous diffusion over the graph.
- The heat diffusion distance compares the diffusion profiles starting from nodes i and j :

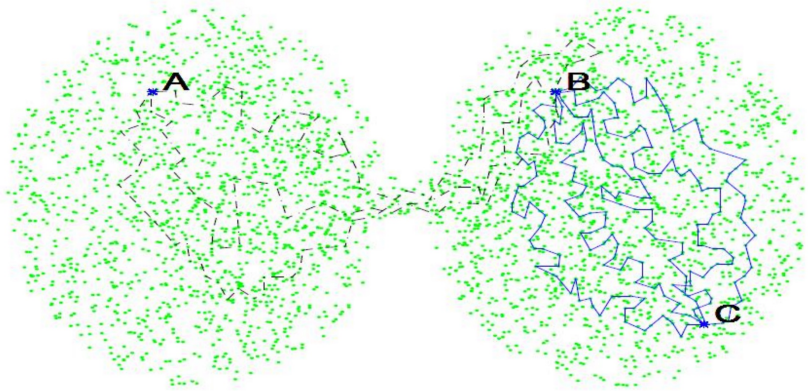
$$d_t^{\text{heat}}(i,j)^2 = \|H_t(i,\cdot) - H_t(j,\cdot)\|_2^2$$
$$= \sum_{\ell=1}^{n} e^{-2t\mu_\ell} \left[\phi_\ell(i) - \phi_\ell(j)\right]^2, \text{ with } L\phi_\ell = \mu_\ell \phi_\ell$$

- Interpretation : The diffusion time t controls the structural scale being explored.
  - Small t emphasizes local neighborhood structure.
  - Larger t suppresses local variations and emphasizes global structure/community.

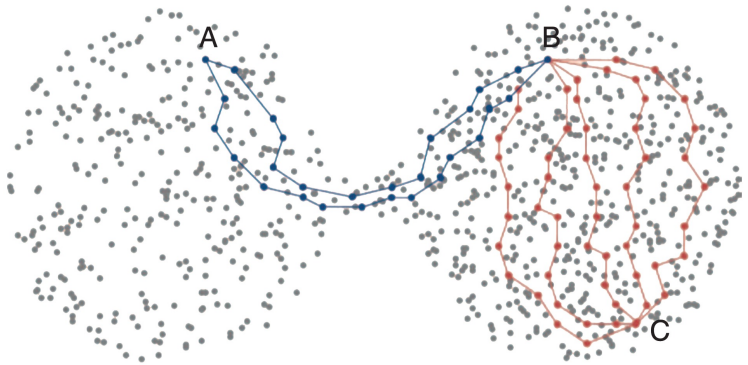
[1] Kondor, Lafferty, Diffusion Kernels on Graphs and Other Discrete Input Spaces, 2002

Diffusion vs. shortest path

- The two distances capture fundamentally different notions of proximity:
  - Shortest-path distance[1] measures geodesic proximity via the shortest connecting path.
  - Diffusion distance[2,3] measures connectivity similarity via random-walk propagation on graph.
- Here, $d^{SP}(A,B) \approx d^{SP}(B,C)$, the shortest connecting paths have comparable lengths despite the different community structure.
- But $d_t^{diff}(A,B) \gg d_t^{diff}(B,C)$, because $P_t(B,\cdot) \approx P_t(C,\cdot)$, while $P_t(A,\cdot)$ differs strongly from $P_t(B,\cdot)$.



$$d_t^{\text{diff}}(A,B) \gg d_t^{\text{diff}}(B,C)$$



$$d^{\text{SP}}(A,B) \approx d^{\text{SP}}(B,C)$$

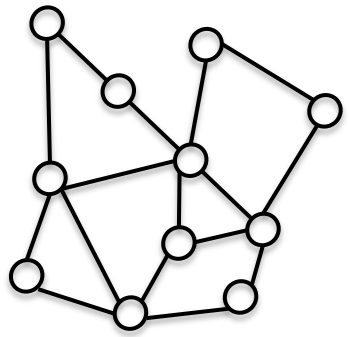
[1] Dijkstra, A Note on Two Problems in Connexion with Graphs, 1959
[2] Talmon, Cohen, Gannot, Coifman, Diffusion Maps for Signal Processing, 2013
[3] Lafon, Diffusion Maps and Geometric Harmonics, 2004

Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Graph-level features

- Graphs vary in size and node ordering : how can we compare them?
- Construct a fixed-size, permutation-invariant representation that summarizes the graph structure.
- Graph feature maps and kernels compare graphs through structural patterns and define graph similarities.
  - Graphlet kernels[1] : Compare graphs using counts of small subgraph patterns.
  - Weisfeiler-Lehman kernels[2] : Compare graphs using iteratively refined node-label patterns.



$\mathcal{G}$

Graph feature representation

$\Rightarrow$

$$h_{\mathcal{G}} = \begin{bmatrix} \\ \end{bmatrix} \in \mathbb{R}^d$$

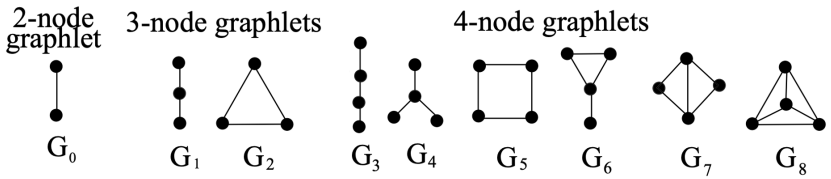
[1] Shervashidze, Vishwanathan, Petri, Mehlhorn, Borgwardt, Efficient graphlet kernels for large graph comparison, 2009
[2] Shervashidze, Schweitzer, Van Leeuwen, Mehlhorn, Borgwardt, Weisfeiler-lehman graph kernels, 2011

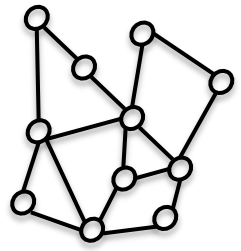
Graphlet feature vector

- Choose the set of k-size graphlets : $S_k = \{G_1,...,G_m\}$, $|G_j| = k$, $\forall j$.
- For each graph $G = (V,E)$, count how many times every graphlet occurs : $c_j(G) = \#\{G_j \subseteq G\}$.
- This gives a graph-level count feature vector $c(G) \in N^m$.
- After normalizing the counts, we obtain a graph-level feature vector $\phi(G) \in R^m$ representing the graphlet frequency distribution of the graph[1] :

$$\phi(G) = \begin{bmatrix} \phi_1(G) \\ \vdots \\ \phi_m(G) \end{bmatrix} \in \mathbb{R}^m, \ \phi_j(G) = \frac{c_j(G)}{\binom{|G|}{|G_j|}} = \frac{c_j(G)}{\binom{|G|}{k}},$$

where $\binom{|G|}{k} = \frac{|G|!}{k!\,(|G|-k)!}$ is the number of possible subsets of $|G_j| = k$ nodes in G.





G

[1] Shervashidze, Vishwanathan, Petri, Mehlhorn, Borgwardt, Efficient graphlet kernels for large graph comparison, 2009

Graphlet kernel

- Given the m-dimensional graphlet feature vector $\phi(G)$ for a graph, two graphs G_i and G_j can be compared using the normalized graphlet kernel K :

$$K(G_i, G_j) = K_{ij} = \frac{\phi(G_i)^T \phi(G_j)}{\|\phi(G_i)\|_2 \, \|\phi(G_j)\|_2} \in [0, 1]$$
$$K = \Phi\Phi^T \in \mathbb{R}^{N_G \times N_G}, \; \Phi \in \mathbb{R}^{N_G \times m}$$

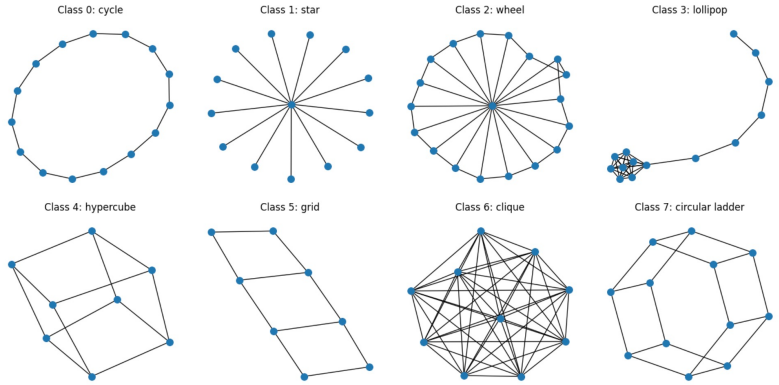
- The resulting kernel matrix can be used with kernel methods s.a. SVMs for graph classification.
- Computational bottleneck : Brute-force counting size-k graphlets in a graph of size N has a complexity of $O(N^k)$, making this method impractical for large values of k.

$$\binom{|G|}{k} = \frac{|G|!}{k!\,(|G|-k)!} = O(|G|^k) \; \rightarrow \; \binom{N}{k} = O(N^k)$$

[1] Shervashidze, Vishwanathan, Petri, Mehlhorn, Borgwardt, Efficient graphlet kernels for large graph comparison, 2009

Lab 2 : Graph classification with Graphlets

- **Implement graph classification with the graphlet kernel :**



Lecture : Shallow Graph Learning

Lab 02 : Graph Classification with Graphlet Kernel -- Solution

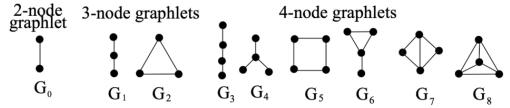
Xavier Bresson

In this notebook we classify the 8 graph classes of DGL's `MiniGCDataset` using the **9 connected graphlets on 2, 3, and 4 nodes**:

$$1 + 2 + 6 = 9.$$

The graphlets are

$$G_0, \dots, G_8 = \{\text{edge},\ P_3,\ K_3,\ P_4,\ K_{1,3},\ C_4,\ \text{tailed triangle},\ K_4 - \text{edge},\ K_4\}.$$



For each graph G, let $c_j(G)$ be the number of induced occurrences of graphlet G_j.

Because the graphlets have different sizes, we normalize every count by the number of possible node subsets of the corresponding size:

$$\phi_j(G) = \frac{c_j(G)}{\binom{|G|}{|G_j|}},$$

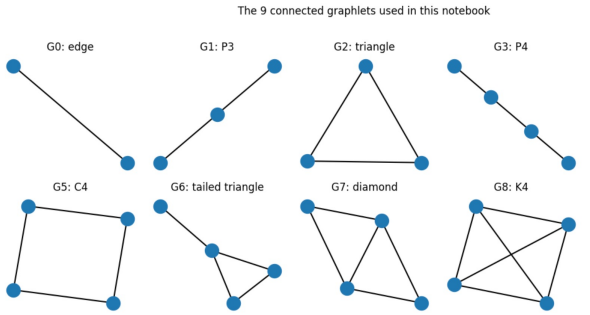
where $\binom{p}{q}$ is the number of ways to choose q elements from a set of p elements, $|G|$ is the number of nodes in the full graph G, and $|G_j|$ is the number of nodes in graphlet G_j.

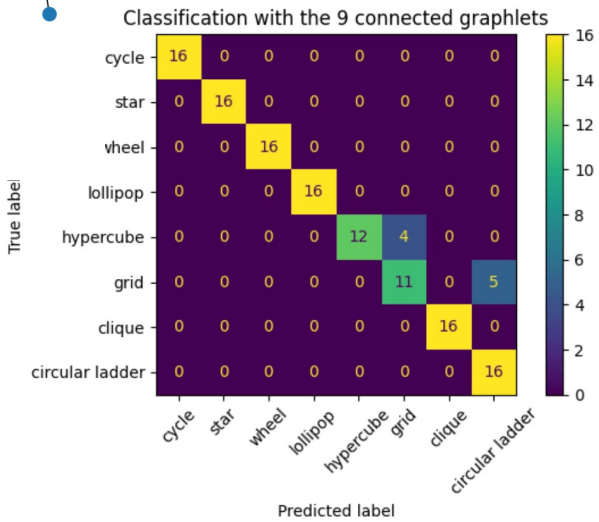
Thus, each graph is represented by a fixed **9-dimensional graphlet feature vector**:

$$\phi(G) = [\phi_0(G), \dots, \phi_8(G)]^\top \in \mathbb{N}^9.$$

We then define the normalized **graphlet kernel**:

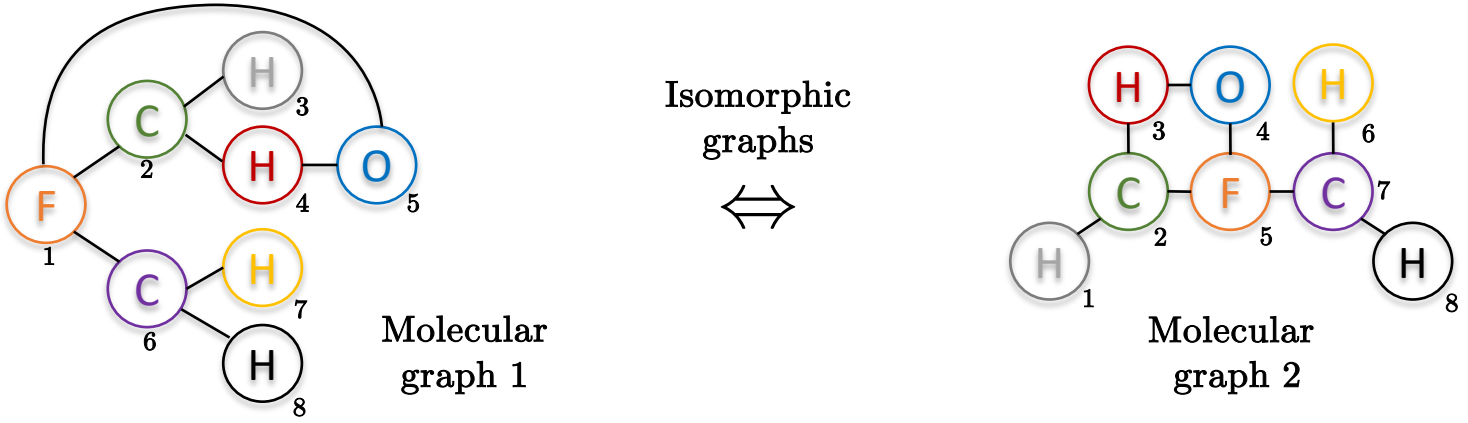
$$K(G, G') = \frac{\phi(G)^\top \phi(G')}{\|\phi(G)\|_2 \, \|\phi(G')\|_2} \in [0, 1]$$





Graph isomorphism

- Graph isomorphism : Two graphs are isomorphic if there is an index permutation between nodes that preserves node adjacencies, i.e. $(u,v) \in E_{G1} \Leftrightarrow (\pi(u),\pi(v)) \in E_{G2}$, for a bijection $\pi : V_{G1} \rightarrow V_{G2}$.
- An example of two isomorphic molecular graphs, node colors indicate the corresponding nodes.
- Determining whether two graphs are isomorphic is NP-intermediate : It is not known if a polynomial time algorithm exists, or the problem is NP-hard.
- A graph-level representation should be invariant to node permutations : Isomorphic graphs must receive the same representation.
- Weisfeiler-Lehman (WL) proposed a fast algorithm[1] to test if two graphs are not isomorphic.

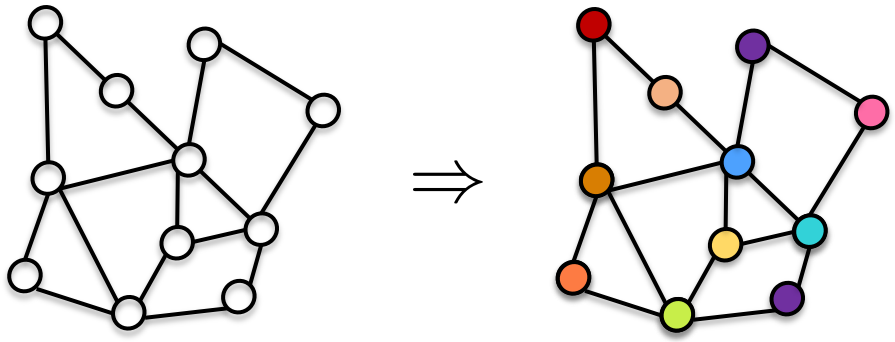


[1] Weisfeiler, Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968

Weisfeiler-Lehman algorithm

- The Weisfeiler-Lehman algorithm[1,2] is an interative coloring process of the nodes of a graph.
- At each iteration, a node color is updated from its current color and the multiset of its neighbors' colors.
- The refinement is repeated until the node coloring stabilizes (i.e. no new color can be created).
- The resulting color histogram defines a representation of the graph structure.
- Mathematically, the coloring function f maps a (node color, neighborhood color multiset) pair to a new node color as follows :

$$
\begin{aligned}
c_i^{t+1} &= f_{\text{WL-coloring}}(c_i^t, \{c_j^t\}_{j \in \mathcal{N}_i}) \in \mathcal{C} = \{1, ..., n_{\text{color}}\} \\
&= f_{\text{Hash}}(c_i^t, \{c_j^t\}_{j \in \mathcal{N}_i}) \in \mathcal{C} \\
h_{\mathcal{G},c} &= \#\{i : c_t^{t=T} = c\}, \ c \in \mathcal{C}, \text{ and} \\
& \qquad T \text{ is the final WL iteration}
\end{aligned}
$$



[1] Weisfeiler, Lehman, A reduction of a graph to a canonical form and an algebra arising during this reduction, 1968
[2] Shervashidze, Schweitzer, Van Leeuwen, Mehlhorn, Borgwardt, Weisfeiler-lehman graph kernels, 2011

Coloring property

- The coloring function is injective, $c_1 \neq c_2 \Rightarrow f(c_1) \neq f(c_2)$, w.r.t. the node–neighborhood colors :

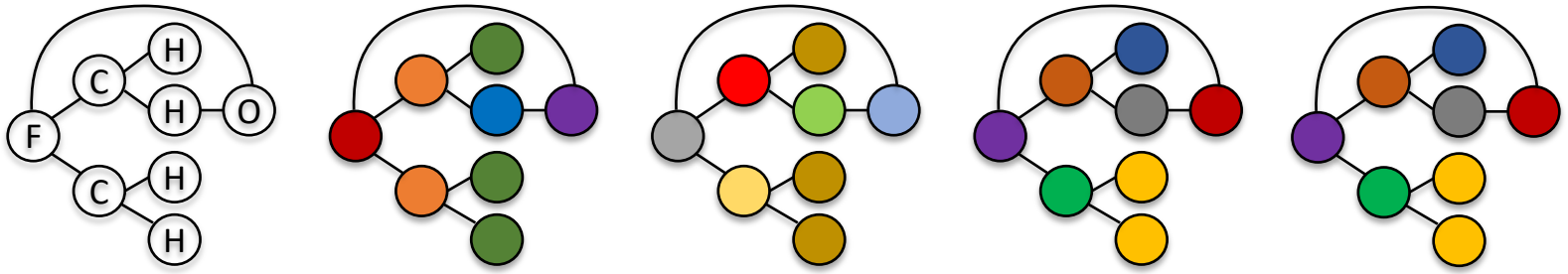
$$f_{\text{WL-coloring}}\left(c_i^t, \{c_j^t\}_{j \in \mathcal{N}_i}\right) = c_i^{t+1}$$

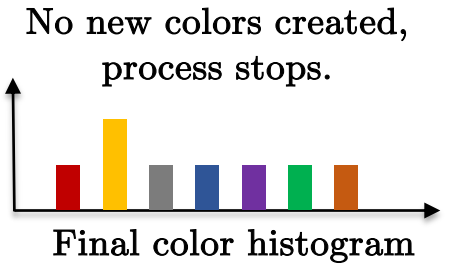
iteration

Multiset (set of unordered and repeated elements)

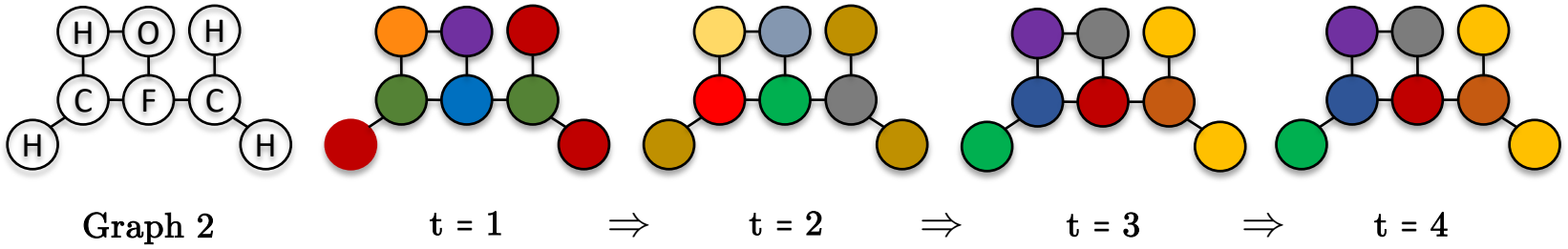
- Consequence of f_{WL} injectivy :
  - Identical (node color, neighborhood color multiset) pairs receive the same color.
  - Distinct (node color, neighborhood color multiset) pairs receive different colors.

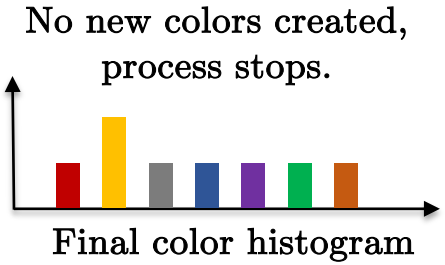
Example





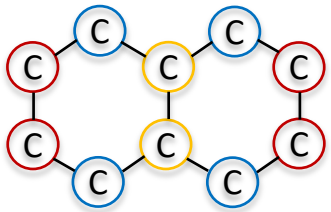
These 2 graphs are
isomorphic/indistinguishable
w.r.t. the WL test.



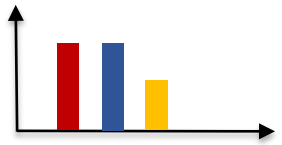


Counter-example

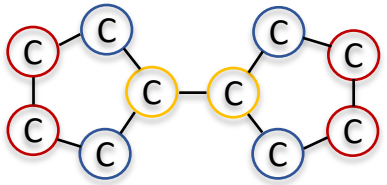
- Are the two graphs below considered isomorphic according to the Weisfeiler–Lehman test ?
- Yes, according to the WL test, the two graphs appear to be isomorphic.
- However, it is clear that they are not actually isomorphic, demonstrating that the WL test provides a necessary but not sufficient condition for graph isomorphism.



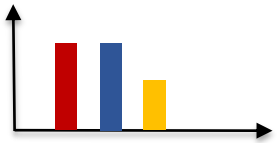
Graph 1
(Decalin)



Histogram of colors



Graph 2
(Bicyclopentyl)



Histogram of colors

Weisfeiler-Lehman kernel

- This produces a permutation-invariant representation of a graph from its color histograms, which can be used to compare graphs and build a kernel[1] for graph classification.
- We build the WL features as follows :
  - At iteration t, let L_t denote the set of WL color labels.
  - For each color label $\ell \in L_t$, we can count the nodes in G with color ℓ at iteration t :

$$h_t(G)_\ell = |\{v \in V : c_t(v) = \ell\}|$$

  - The WL feature vector is obtained by concatenating the color-count histograms from all iterations ($N_c = \Sigma_{t=0}^{T}|L_t|$ is the number of created colors) :

$$\phi_T(G) = [h_0(G), h_1(G), \cdots, h_T(G)]^T \in \mathbb{R}^{N_c}$$

- The WL kernel between two graphs is then defined as :

$$K = \Phi_T^T \Phi_T \in \mathbb{R}^{N_G \times N_G}, \ \Phi_T \in \mathbb{R}^{N_c \times N_G}$$

[1] Shervashidze, Schweitzer, Van Leeuwen, Mehlhorn, Borgwardt, Weisfeiler-lehman graph kernels, 2011

Lab 3 : Graph classification with WL

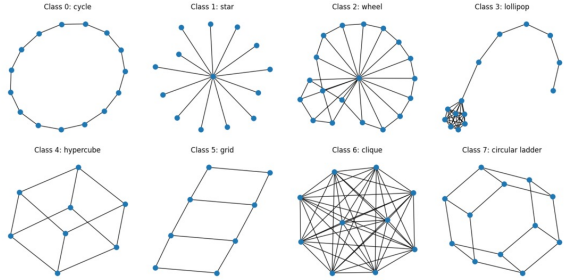
- Implement graph classification with WL kernel.

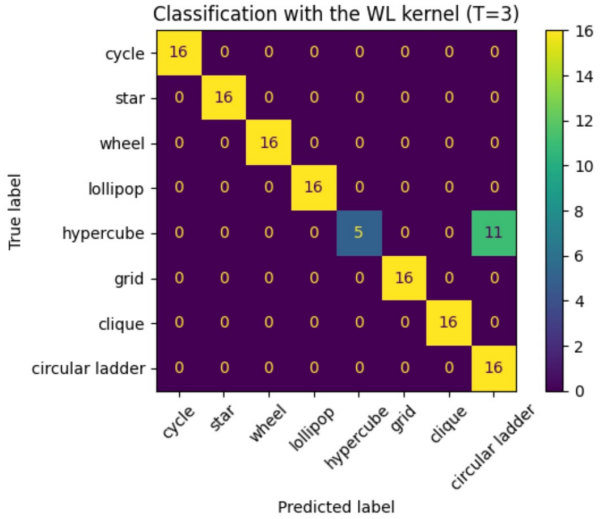
Lecture : Shallow Graph Learning

Lab 03 : Graph Classification with Weisfeiler–Lehman Kernel -- Solution

Xavier Bresson

In this notebook we classify the 8 graph classes of DGL's `MiniGCDataset` using the **Weisfeiler–Lehman (1-WL) kernel**.





The graphs in `MiniGCDataset` do not have semantic node labels, so every node starts with the same color label:

$$c_0(v) = 0, \ \forall v \in V.$$

At the WL iteration t, each node v is relabeled from its current color label and the multiset of color labels in its one-hop neighborhood:

$$c_t(v) = \text{HASH}\left(c_{t-1}(v), \{c_{t-1}(u) : u \in N(v)\}\right) \in \mathbb{R},$$

where HASH is an injective function that takes as input a node's current color and the multiset of its neighbors' colors, and outputs a new color label. Thus, two nodes receive the same new color if and only if these two signatures are identical.

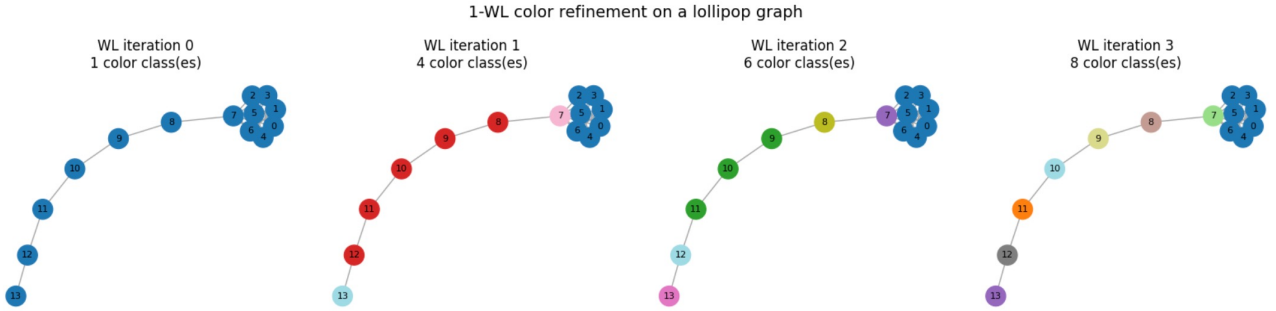
For each iteration $t = 0, \ldots, T$, we count how many nodes receive each WL label. Concatenating these histograms gives a graph-level feature vector:

$$\phi_T(G) = [c_0(G), c_1(G), \ldots, c_T(G)]^\top \in \mathbb{R}^T.$$

The **WL kernel** is the normalized inner product:

$$K_T(G, G') = \frac{\phi_T(G)^\top \phi_T(G')}{\|\phi_T(G)\|_2 \, \|\phi_T(G')\|_2} \in [0, 1]$$

and train a kernel SVM model.

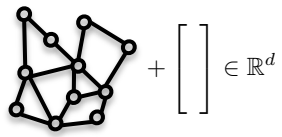


Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- **Shallow graph feature learning**
  - **Node feature**
  - Edge & graph feature
- Conclusion

Graph machine learning

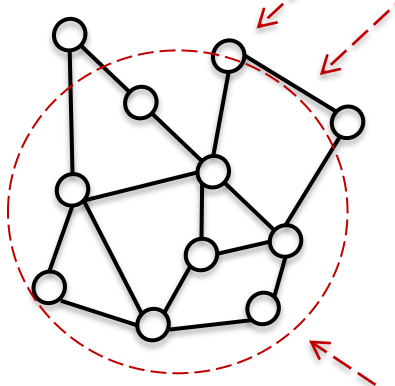
- The graph machine learning pipeline :



$+ \begin{bmatrix} \\ \end{bmatrix} \in \mathbb{R}^d$

1) Graph topology
2) Node-, edge-, and graph-level features

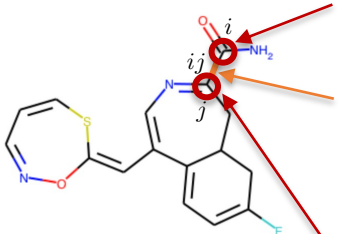
Input $\Rightarrow$ Graph Machine Learning (GML) $\Rightarrow$ Output/ prediction



Node-level prediction

Edge-level prediction

Graph-level prediction



Quantum Chemistry

$\text{Atom}_i \begin{bmatrix} \text{type} \\ \text{coordinates} \\ \text{charge} \end{bmatrix}_i \in \mathbb{R}^{d_v}$

Bond_{ij} $A_{ij} = \begin{cases} 1 & \text{if ij bond} \\ 0 & \text{otherwise} \end{cases}$

$\begin{bmatrix} \text{type} \\ \text{energy} \end{bmatrix}_{ij} \in \mathbb{R}^{d_e}$

$\text{Atom}_j \begin{bmatrix} \text{type} \\ \text{coordinates} \\ \text{charge} \end{bmatrix}_j \in \mathbb{R}^{d_v}$

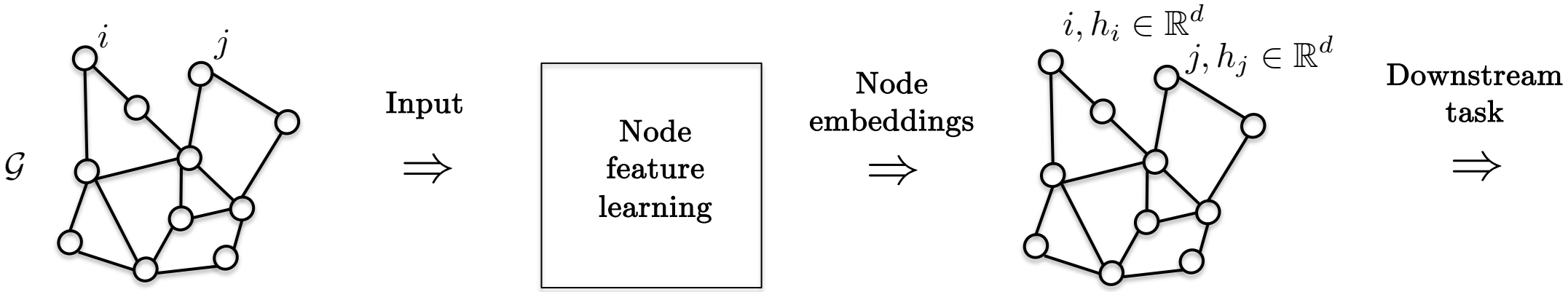
$\Rightarrow$ Graph feature learning $\Rightarrow$ Downstream task

GML

$\Rightarrow$ Molecular energy

Node feature learning

- In this lecture, we focus on the node feature learning task :

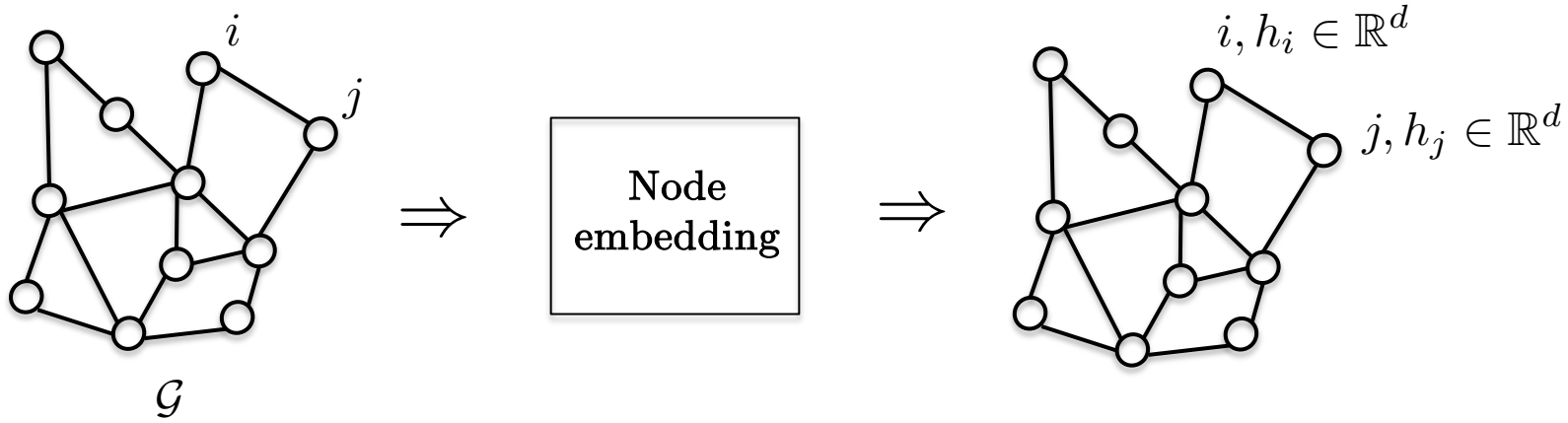


1) Graph topology
2) Node-level features

- Terminology : "Node representation learning" is commonly used interchangeably with "node feature learning". More generally, representation learning can be performed at the node, edge, or graph level.

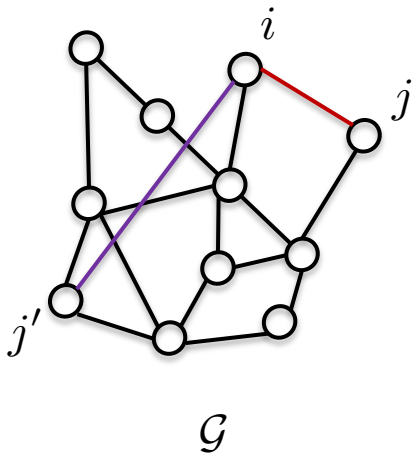
Shallow node feature learning

- Hypothesis
  - Shallow learning : We use a single learning layer to assign a vector to each node, a.k.a. node embedding / encoding layer.
  - The embeddings are learned from graph topology, typically encoded by the graph adjacency matrix.
  - No external node-level attributes are used, e.g. no atom types nor other input features.
- How do we learn node embedding vectors $h_i \in R^d$, $i \in V$, from the graph topology only?



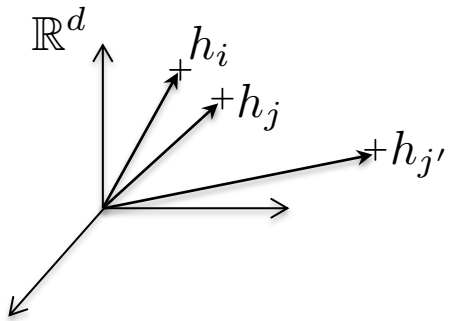
Fundamental embedding property

- The embedding space must preserve the proximity of nodes.
- If nodes i and j are close on the graph G, their corresponding embedding vectors (h_i,h_j) in R^d should also be close in the embedding space.



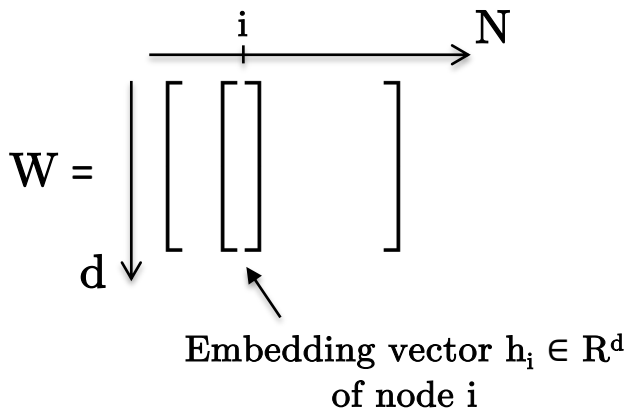
Node embedding

$\Rightarrow$



Shallow embedding

- The most straightforward shallow node feature can be constructed as follows :
  - One-hot encoding : Each node index i ∈ {0,1,...,N-1} is represented by the basis vector $e_i \in R^N$, a.k.a. as one-hot encoding vector e_i = [0,0,..0,1,0,...,0] ∈ R^N.
  - Linear Transformation : This one-hot vector is then linearly transformed to produce the node embedding vector $h_i = We_i \in R^d$, with $W \in R^{d \times N}$.
- Matrix W is also called an Embedding lookup, which maps node i to $h_i = We_i = W_{.,i} \in R^d$.
- Interpretation : W is a learnable embedding table whose columns are optimized using a graph-based objective to capture node relationships.

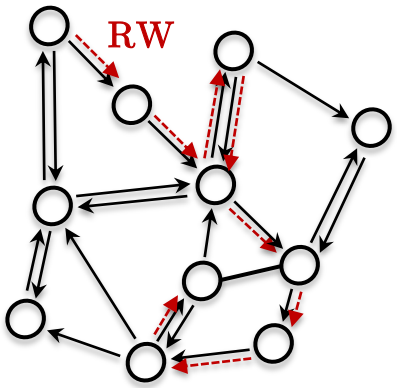


Learning pipeline

- How do we learn the embedding parameters W ?
- Standard learning procedure
  - Select a similarity metric : Choose a distance measure to evaluate the similarity between two node embeddings.
  - Design a loss function : Create a loss function that promotes embedding similarity based on the selected metric.
  - Optimize parameters : Use gradient descent optimization to find the optimal parameters W that minimize the loss function.

Node-to-vector embedding

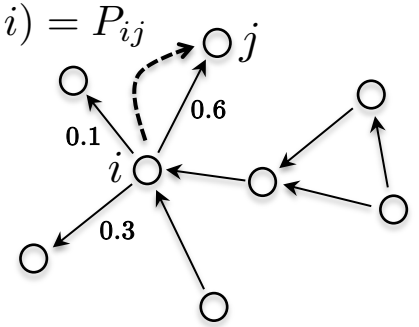
- Motivation : Adapt word2vec[1] / skip-gram from word sequences to graphs.
  - DeepWalk[2] : uniform random walks
  - Node2vec[3] : biased random walks
- Key idea :
  - Generate random walk (RW) sequences of nodes, treating nodes as words and walks as sentences.
  - Learn embeddings so that nodes co-occurring within a walk context have similar representations.



[1] Mikolov, Sutskever, Chen, Corrado, Dean, Distributed representations of words and phrases and their compositionality, 2013
[2] Perozzi, Al-Rfou, Skiena, Deepwalk: Online learning of social representations, 2014
[3] Grover, Leskovec, node2vec: Scalable feature learning for networks, 2016

Random walk generation

$$\Pr_{i \to j} = \Pr(j|i) = P_{ij}$$



- How do we generate a random walk ?
- Define the random walk probability transition matrix as :

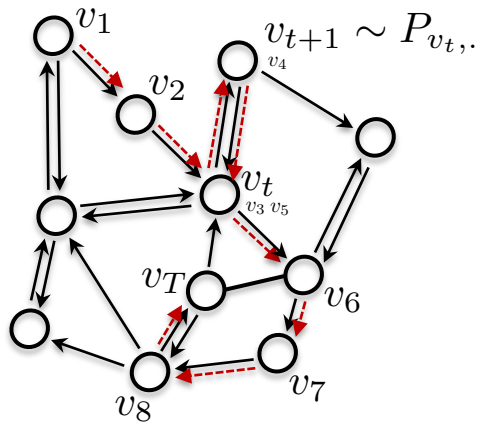
$$P = D^{-1}A \in \mathbb{R}^{N \times N}, \quad D_{ii} = \sum_j A_{ij}$$

- Each row of P, i.e. $P_{i,.}$, gives the probability to move from vertex i to vertex j, with $\Sigma_j P_{ij} = 1$.
- A random walk is generated by repeatedly sampling $v_{t+1} \sim$ Bernouilli/Categorical($P_{vt,.}$),

  set $v_t \leftarrow v_{t+1}$ and repeat for the desired number T of steps.
- The probability of a random-walk trajectory is

$$P_{\mathrm{RW}}(v_1, \ldots, v_T) = P(v_1) \prod_{t=1}^{T-1} P(v_{t+1}|v_t)$$

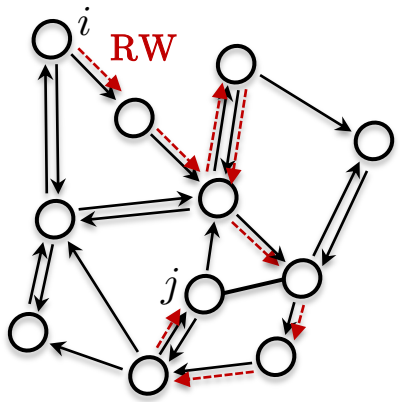
where the starting node is uniformly sampled $v_1 \sim P(v_1) = 1/|V|$



Sampling a RW path (⇢) in a graph

Random walk property

- Random walk exploration is stochastic, which provides both advantages and limitations.
- Advantages : RWs are inexpensive to sample and can be generated in parallel.
- Limitations : Many walks may be required for sufficient graph coverage, since finite walks are biases as they typically explore local neighborhoods.
- Overall, RWs identify nodes that are closely connected in the graph.
- We will assume that two nodes i and j are close if they belong to the same RW.



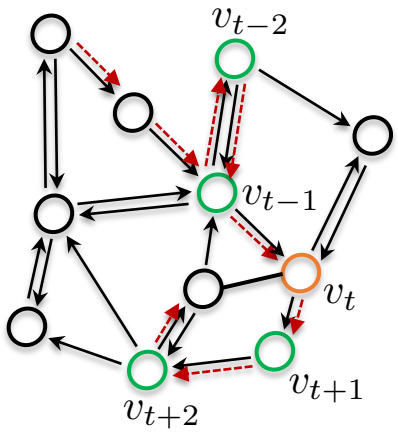
RW starting at i and reaching j after a few steps/hops.

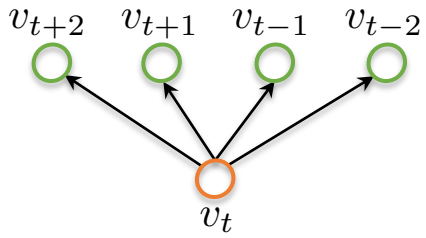
Skip-gram model

- After a random walk is generated, choose node v_t as a center node and define its context $C_t = C(v_t)$ using a window around node t.
- Skip-gram assumes that context nodes C_t are conditionally independent given the center node v_t :

$$P(C_t|v_t) = \prod_{j \in C_t} P(j|v_t)$$

- Thus, skip-gram model treats center node v_t as a center word and the nodes in C_t as context words to be predicted from v_t.





$P(j|v_t),\ j \in C_t$

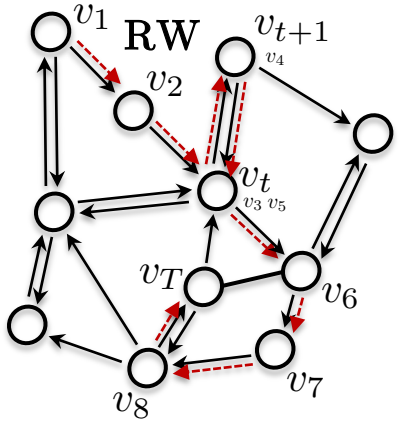
with $C_t = \{v_{t-2}, v_{t-1}, v_{t+1}, v_{t+2}\}$

Probability of skip-gram model

- The model learns to predict each context node $j \in C_t$ from the center node v_t.
- For a sampled random walk $RW = (v_1,\ldots,v_T)$, the skip-gram likelihood of all observed center–context pairs is defined as

$$P_{\mathrm{SG}}(\mathrm{RW}) = \prod_{t=1}^{T} \prod_{j \in C_t} P(j|v_t)$$

$$\log P_{\mathrm{SG}}(\mathrm{RW}) = \sum_{t=1}^{T} \sum_{j \in C_t} \log P(j|v_t)$$



where the log function is used for faster and more stable computations.

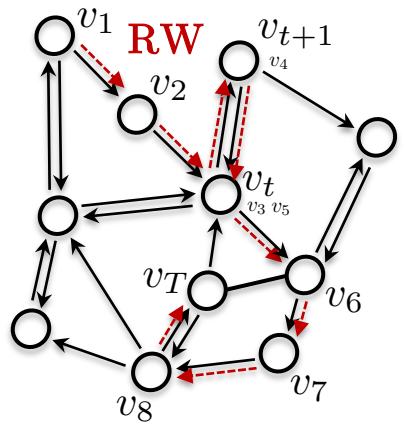
Learning skip-gram node embeddings

- To learn node embeddings that capture graph topology, we will maximize the expected skip-gram log-likelihood over random walks sampled from the graph G :

$$\max_{W} \; \mathbb{E}_{R \sim P_{\mathrm{RW}}(\cdot|G)} \left[\sum_{t=1}^{T} \sum_{j \in C_t} \log P_W(j|v_t) \right]$$

$W \in \mathbb{R}^{d \times N}$ denotes the learnable node embedding parameters.

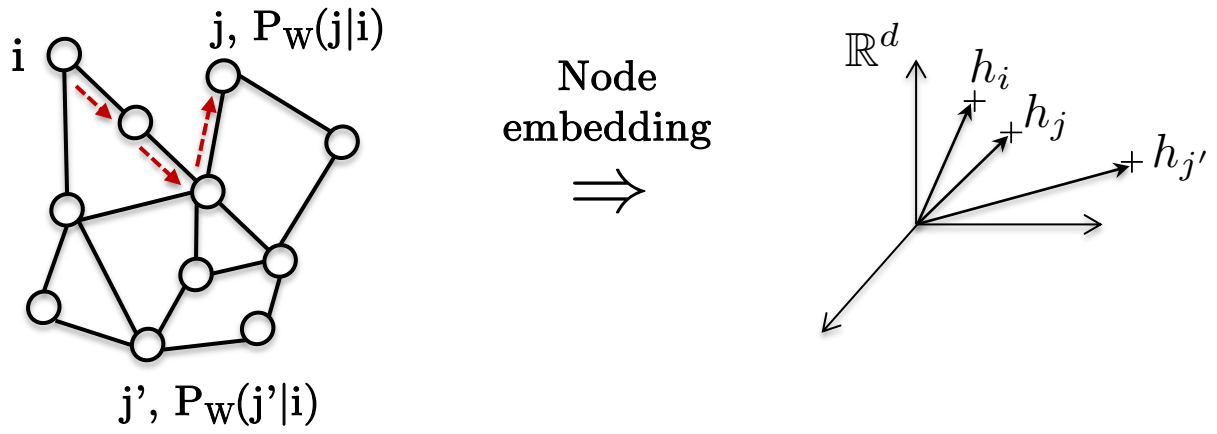
Expectation/average over sampled random walks.



- Maximizing this objective enforces nodes that co-occur in random walk contexts to have similar embeddings, thereby preserving graph proximity and higher-order connectivity.

From graph proximity to conditional probability

- How to parametrize the conditional conditional $P_W(j|i)$ using node embeddings h_i, $h_j \in R^d$?
  - If nodes i and j are close on the graph (equivalently in a sampled RW), then their embeddings $(h_i,h_j) \in R^d$ should have a high similarity score.
  - Equivalently, if nodes i and j are close on the graph, their conditional probability $P_W(j|i)$ should have a high value.
  - Conclusion : High probability $P_W(j|i)$ implies similar embeddings (h_i,h_j), and inversely.



Parameterizing skip-gram probability

- **Different parameterizations of $P_W(j|i)$ are possible :**

$$P_W(j|i) = \frac{\sigma(s_{ij})}{Z_i} \qquad \text{Conditional probability}$$

$$\sigma(s) = \begin{cases} \exp(s) \\ \text{sigmoid}(s) = (1+e^{-s})^{-1} \end{cases} \qquad \text{Base function (sparse/dense selection)}$$

$$s_{ij} = \begin{cases} h_i^T h_j \\ -\|h_i - h_j\|_2^2 \end{cases} \qquad \text{Embedding similarity score}$$

$$Z_i = \sum_{j' \in V} \exp(s_{ij'}) \qquad \text{Normalization constant (partition function)}$$

$$h_n = We_n, \in \mathbb{R}^d, n \in V \qquad \text{Learnable node embedding}$$

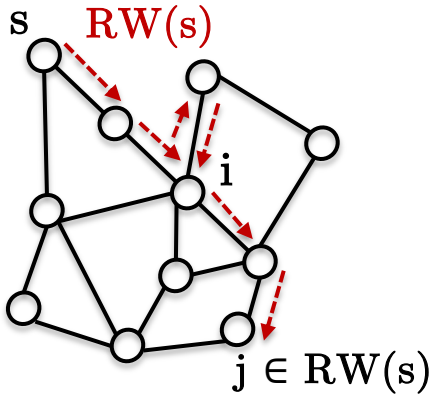
Sampling-based approximation

- We approximate the expected skip-gram loss over random walks sampled from G.
- Sampling-based approximation :
  - Starting nodes : Use nodes s in V as starting points for random walks.
  - RW generation : For each starting node s, sample one or more fixed-length random walks RW(s) (easily parallelized).
  - Context extraction : For each occurrence v_t in RW(s), define context C_t using a local window around v_t.
- Objective : Minimize/max the empirical negative skip-gram loss over the sampled walks :

$$\min_W \; L(W) = -\sum_{s \in V} \sum_t \sum_{j \in C_t(s)} \log P_W(j|v_t(s))$$

$$\Updownarrow$$

$$\max_W \sum_{s \in V} \sum_t \sum_{j \in C_t(s)} \log P_W(j|v_t(s))$$



Brute-force training

- Two-step training algorithm :
  - Random walk sampling : For each node s in V, sample one or more fixed-length random walks RW(s) (walks can be generated in parallel).
  - Loss maximization : Extract center–context pairs from the sampled walks and maximize the negative skip-gram log-likelihood :

$$\max_W \sum_{s \in V} \sum_t \sum_{j \in C_t(s)} \log P_W(j|v_t(s))$$

$$\text{with } P_W(j|i) = \frac{\exp(h_i^T h_j)}{\sum_{j' \in V} \exp(h_i^T h_{j'})}, \; h_n = W e_n \in \mathbb{R}^d$$

  - However, the full softmax is expensive, making the overall complexity quadratic, i.e. $O(|V|^2) = O(N^2)$.

Negative sampling

- It is a fast technique[1] for avoiding the expensive computation of the partition function Z (denominator) in the probability, by replacing multi-class prediction with binary classification.
- The goal is to approximate the original problem by considering nodes outside the RW path : For each positive center–context pair (v_t, $j \in C_t$), sample K negative nodes $k_1,\ldots,k_K \sim q(k)$ (usually uniform distribution over V), and optimize :

$$\max_W \sum_{s \in V} \sum_t \sum_{j \in C_t(s)} \log P_W(j|v_t(s))$$

$\Downarrow$ approximate with negative sampling

$$\max_W \sum_{s \in V} \sum_t \sum_{j \in C_t(s)} \sum_{r=1}^{K} \log \frac{P_W(j|v_t(s))}{P_W(k_r|v_t(s))}$$

Positive samples

Negative samples

- Maximizing the new objective reinforces the probability of positive samples (as maximizing ratio $P_{pos}/P_{neg} \Rightarrow P_{pos} > P_{neg}$).

[1] Mikolov, Sutskever, Chen, Corrado, Dean, Distributed representations of words and phrases and their compositionality, 2013

Key insight

- **The partition function Z can be discarded (by replacing sum_V with $\text{sum}_{\text{K}\ll\text{V}}$) :**

$$\max_W \; L_{NS}(W) = \sum_{s\in V}\sum_{t}\sum_{j\in C_t(s)}\sum_{r=1}^{K}\log\frac{P_W(j|v_t(s))}{P_W(k_r|v_t(s))}$$

$$= \sum_{s\in V}\sum_{t}\sum_{j\in C_t(s)}\sum_{r=1}^{K}\log\frac{\sigma(s_{v_t(s)j})/Z_{v_t(s)}}{\sigma(s_{v_t(s)k_r})/Z_{v_t(s)}} \quad \text{with } P_W(j|i)=\frac{\sigma(s_{ij})}{Z_i},\; Z_i=\sum_{j'\in V}\sigma(s_{ij'})$$

$$= \sum_{s\in V}\sum_{t}\sum_{j\in C_t(s)}\left(\log\sigma(s_{v_{st}j})-\sum_{r=1}^{K}\log\sigma(s_{v_{st}k_r})\right) \quad \text{with } \log(\frac{a}{b})=\log a-\log b$$

$$= \sum_{s\in V}\sum_{t}\sum_{j\in C_t(s)}\left(\log\sigma(h_{v_{st}}^T h_j)-\sum_{r=1}^{K}\log\sigma(h_{v_{st}}^T h_{k_r})\right) \quad \text{with } s_{ij}=h_i^T h_j, h_n=We_n, \in\mathbb{R}^d, n\in V$$

with $h_{v_{st}}^T h_j$ for positive pairs and $h_{v_{st}}^T h_{k_r}$ for negative pairs.

Efficient algorithm

- DeepWalk training[1]
  - Random-walk sampling : For each node s in V, sample one or more fixed-length random walks WR(s) in parallel.
  - Embedding optimization : Extract center–context pairs, and minimize the negative-sampling loss using gradient descent :

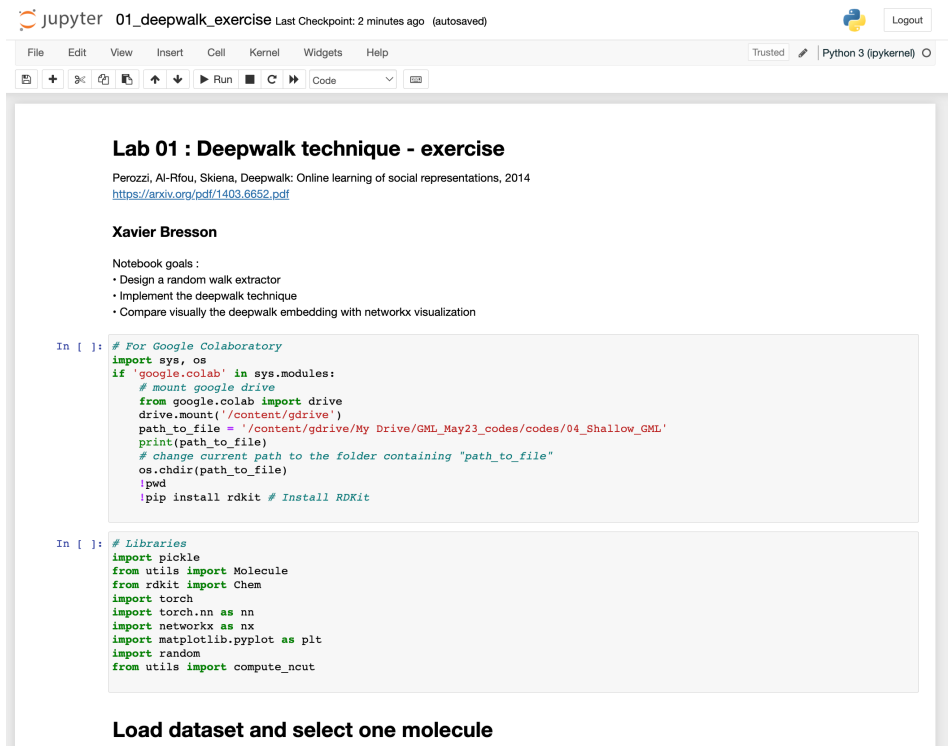
$$\min_W L_{\mathrm{NS}}(W) = -\sum_{s\in V}\sum_{t}\sum_{j\in C_t(s)}\left(\log\sigma(h_{v_{st}}^T h_j) - \sum_{r=1}^{K}\log\sigma(h_{v_{st}}^T h_{k_r})\right),\ k_r\sim q(k)$$

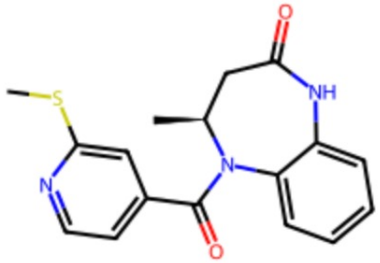
  - The training cost scales as O(N.R.T.w.K.d) = O(N), where N is the number of nodes, R is walks per node, T walk length, w context-window size, K negative samples and d embedding dimension.

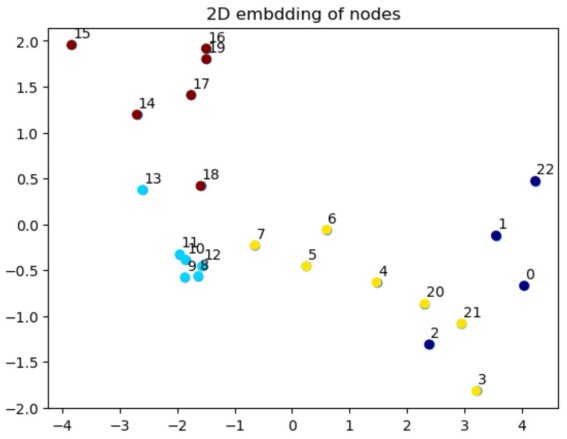
[1] Perozzi, Al-Rfou, Skiena, Deepwalk: Online learning of social representations, 2014

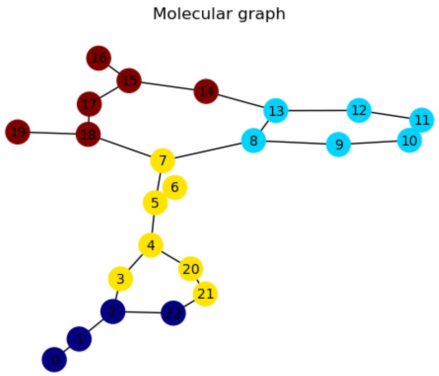
Lab 4 : DeepWalk

- Implement DeepWalk[1] (2024 KDD Test of Time Award)









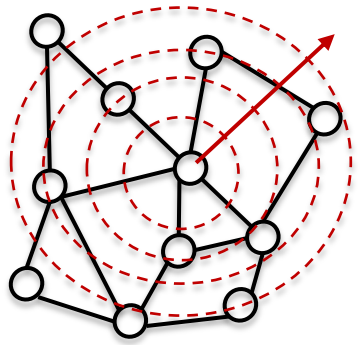
[1] Perozzi, Al-Rfou, Skiena, Deepwalk: Online learning of social representations, 2014

Node2vec : biased random walks

- Standard random walks explore graph neighborhoods using fixed, topology-based transition probabilities (no guidance).
- While this approach is mathematically sound, it requires a large number of RWs to accurately approximate the skip-gram loss:

$$L(W) = \mathbb{E}_R \left[-\sum_t \sum_{j \in C_t} \log P_W(j|v_t) \right]$$

- Node2vec[1] introduces biased random walks, controlled by scalar parameters p and q, to flexibly interpolate between local/BFS-like (breadth-first search) and outward/DFS-like (depth-first search) exploration.



[1] Grover, Leskovec, node2vec: Scalable feature learning for networks, 2016

Node2vec : biased transition probabilities

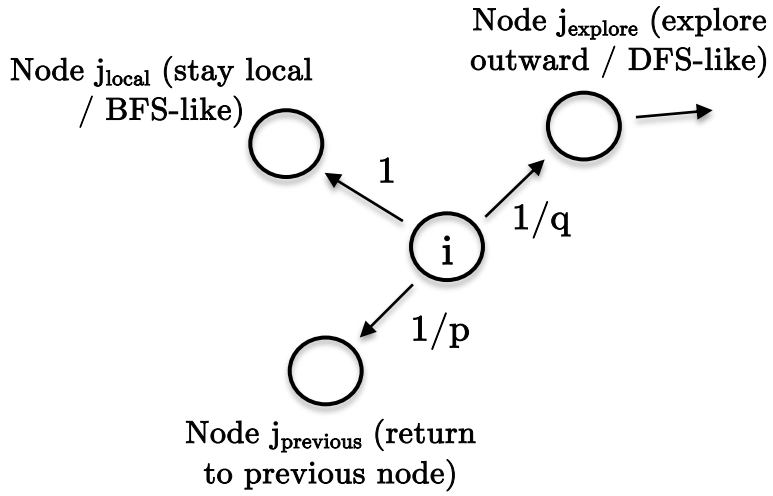
- The new transition rule for random walks is defined as

$$P(v_{t+1} = j \mid v_t = i, v_{t-1} = u) = \frac{\alpha_{p,q}(u, j)}{\sum_{x \in \mathcal{N}(i)} \alpha_{p,q}(u, x)}$$

$$\text{with} \quad \alpha_{p,q}(u, j) = \begin{cases} 1/p, & d(u, j) = 0 \text{ (return to previous node)} \\ 1, & d(u, j) = 1 \text{ (local/BFS-like move)} \\ 1/q, & d(u, j) = 2 \text{ (outward/DFS-like exploration)} \end{cases}$$

- The interpretation of q is :
  - q $>$ 1 ⇒ more local/BFS-like
  - q $<$ 1 ⇒ more outward/DFS-like.

- Node-to-vec maintains a training linear complexity with the graph size, i.e. O(N).



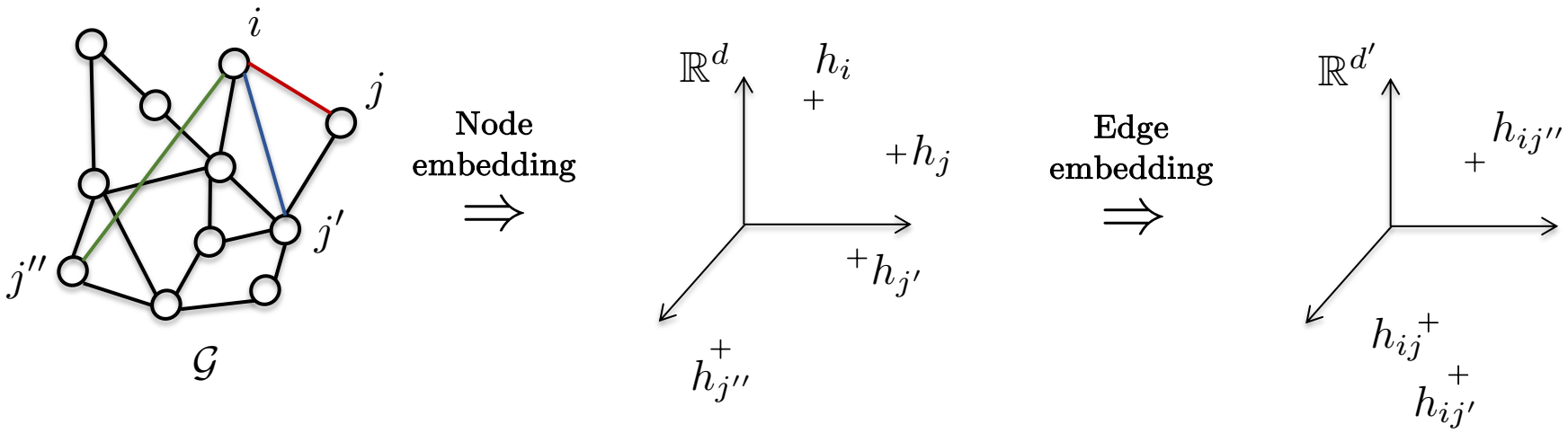
Outline

- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Edge representations from node embeddings

- **Edge representation from node embeddings : Represent a node pair (i,j) by combining their node embeddings.**

$$h_{ij} = h_i + h_j \in \mathbb{R}^d, \quad h_{ij} = h_i \odot h_j \in \mathbb{R}^d, \quad h_{ij} = |h_i - h_j| \in \mathbb{R}_+ \quad \text{(undirected graphs)}$$
$$h_{ij} = \text{Concat}(h_i, h_j) \in \mathbb{R}^{2d}, \; h_{ij} \neq h_{ji} \quad \text{(directed graphs)}$$
$$\text{with } h_n = We_n \in \mathbb{R}^d, \; n \in V \quad \text{(node embedding)}$$



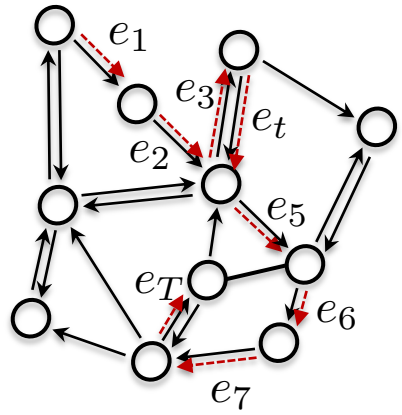
Edge embeddings with random walks

- Edge embeddings can be learned from random-walk edge sequences by treating an edge e = (i,j) as the center and nearby edges e' = (k,l) as its context[1].
- Skip-gram learns similar representations for edges that frequently co-occur within random-walk context windows, thereby capturing structural proximity between links.

$$h_e = W e_e \in \mathbb{R}^d, \qquad W \in \mathbb{R}^{d \times |E|}$$

$$P_W(e'|e) = \frac{\exp(h_e^T h_{e'})}{\sum_{\tilde{e} \in E} \exp(h_e^T h_{\tilde{e}})}$$

$$L(W) = \mathbb{E}_R \left[-\sum_t \sum_{e' \in C_t} \log P_W(e'|e_t) \right]$$



RW

[1] Li et-al, Edge Representation Learning for Community Detection in Large Scale Information Networks, 2018

Graph embeddings with anonymous walks

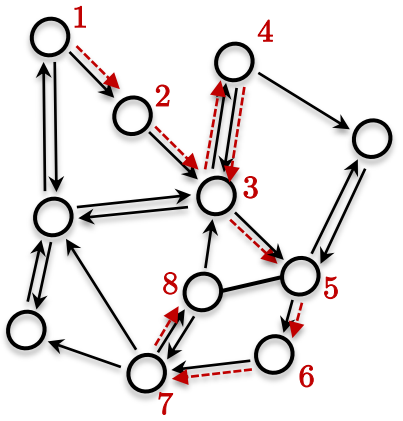
- Learning graph-level embeddings :
  - Anonymous walk embeddings[1] : Sample anonymous walks (AW) from G and learn a graph vector by predicting a target walk from its context walks and the graph representation.
  - Anonymous walk : Replaces node identities by their order of occurrence, retaining only the structural visiting pattern. E.g. $(v_3,v_7,v_3,v_2,v_7) \rightarrow (1,2,1,3,2)$.
  - Permutation invariance : Relabeling the graph nodes does not change the pattern of a anonymous walk, i.e. AWs are invariant to node ordering.
  - The learned graph vector summarizes structural patterns captured by the distribution and co-occurrence of anonymous walks.

$$\max_{\mathbf{h}_G, W} \mathbb{E}[\log P_W(a_t | C_t, h_G)], \; h_G \in \mathbb{R}^d$$

where a_t is a sampled anonymous walk,

C_t is its context of co-occurring anonymous walks,

h_G is the learnable graph representation.



AW=(1,2,3,4,3,5,6,7,8)

[1] Ivanov, Burnaev, Anonymous walk embeddings, 2018

Outline

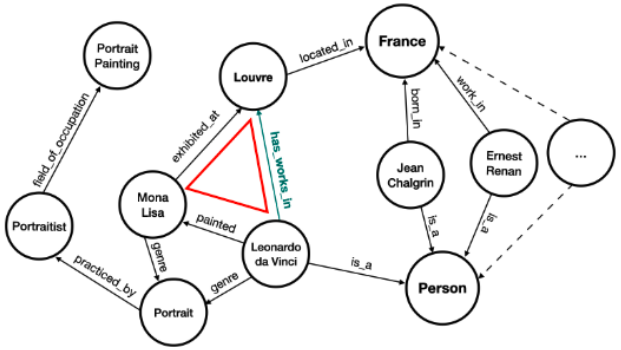
- Graph prediction tasks and features
- Hand-crafted graph feature
  - Node feature
  - Edge feature
  - Graph feature
- Shallow graph feature learning
  - Node feature
  - Edge & graph feature
- Conclusion

Conclusion

- Engineered / hand-crafted graph features
  - Engineered graph features are limited in expressivity and are often domain-specific, relying heavily on expert knowledge.
- Shallow learned features
  - Techniques like DeepWalk / Node2Vec learn node embeddings directly from graph topology and unsupervised losses.

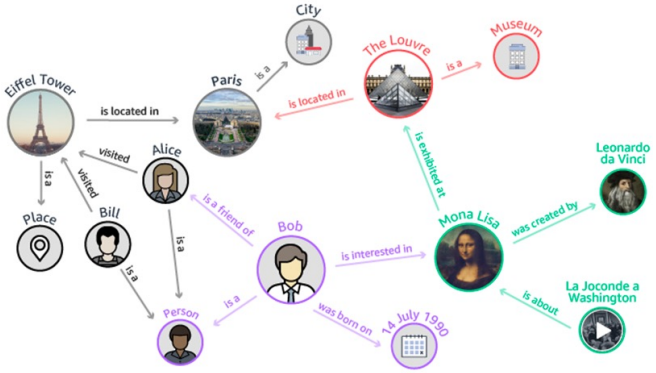
Conclusion

- Transductive learning
  - Can we transfer the node embeddings trained on graph G_1 to a new graph G_2?
  - No, DeepWalk learns a separate embedding vector for each node in the training graph G_1, i.e. $h_i = We_i$, $W \in R^{d \times n1}$.
  - Hence, the embeddings are tied to the specific nodes of G_1.
  - Nodes in a new graph G_2 do not have learned embeddings, unless the model is re-trained.



Source[1]

Transfer?



Source[2]

[1] Hebatallah et-al, Locality-aware subgraphs for inductive link prediction in knowledge graphs, 2023
[2] https://aws.amazon.com/neptune/knowledge-graphs-on-aws

Conclusion

- Limitations of shallow embeddings
  - Limited expressivity : The use of a single learnable layer restricts the model's expressivity and generalization capabilities -- multiple-layer architectures with compositional neighborhood -- aggregation function, are generally more effective.
  - Ignoring raw data features : The methods discussed here rely mainly on topology and do not integrate rich node or edge features, which means critical information are missing.
  - Lack of transferability : Transductive embeddings are tied to nodes seen during training and cannot generalize to unseen nodes or graphs. Hence, a new unseen node has no embedding unless the model is retrained or extended.
- Next step : Graph Neural Networks

Lab 5 : Graph coding warm-up

- Hands-on introduction to graph programming with NetworkX, PyTorch, DGL and PyG.

Lecture : Shallow Graph Learning

Lab 05 : Introduction to Graphs with DGL, PyG, NetworkX, PyTorch - Solution

Xavier Bresson, Guoji Fu

Notebook goals :

- Build graphs with Deep Graph Library (DGL) [1] and PyTorch Geometric (PyG) [2]
- Compute the same message-passing function with DGL and PyG
- Create batches of DGL and PyG graphs
- Convert DGL/PyG graphs to NetworkX graphs for visualization w/ balanced springs system [3]
- Convert DGL/PyG graphs to PyTorch graphs for visualization w/ Laplacian eigenvectors [4]
- Conversions between DGL, PyG, NetworkX, and (dense and sparse) PyTorch graphs

[1] Wang et-al, Deep graph library: A graph-centric, highly-performant package for graph neural networks, 2019
https://www.dgl.ai
[2] Fey, Lenssen, Fast Graph Representation Learning with PyTorch Geometric, 2019
https://pytorch-geometric.readthedocs.io
[3] Kamada, Kawai, An algorithm for drawing general undirected graphs, 1989
https://shorturl.at/fpDV2
[4] Belkin, Niyogi, Laplacian eigenmaps for dimensionality reduction and data representation, 2003
http://graphics.stanford.edu/courses/cs233-20-spring/ReferencedPapers/Laplacian.pdf

```
# For Google Colaboratory
import sys, os
if 'google.colab' in sys.modules:
    # mount google drive
    from google.colab import drive
    drive.mount('/content/gdrive')
    path_to_file = '/content/gdrive/My Drive/CS5284_2026_codes/05_Shallow_Learning'
    print(path_to_file)
    # change current path to the folder containing "path_to_file"
    os.chdir(path_to_file)
    !pwd
    !pip install dgl==1.0.2 # Install DGL
    !pip install torch_geometric==2.5.1 # Install PyG
```

```
# dgl
 Graph(num_nodes=7, num_edges=14,
       ndata_schemes={}
       edata_schemes={}) <class 'dgl.heterograph.DGLGraph'>


# dgl => networkx
 MultiDiGraph with 7 nodes and 14 edges <class 'networkx.classes.multidigraph.MultiDiGraph'>


# networkx => dgl
 Graph(num_nodes=7, num_edges=14,
       ndata_schemes={}
       edata_schemes={}) <class 'dgl.heterograph.DGLGraph'>
Note: Node and edge features must be re-generated!


# networkx => (coo) sparse pytorch
 tensor(indices=tensor([[0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6],
                        [1, 6, 0, 2, 1, 3, 2, 4, 3, 5, 4, 6, 0, 5]]),
        values=tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]),
        size=(7, 7), nnz=14, layout=torch.sparse_coo) <class 'torch.Tensor'>


# (coo) sparse pytorch => networkx
 Graph with 7 nodes and 7 edges <class 'networkx.classes.graph.Graph'>


# (coo) sparse pytorch => dense pytorch
 tensor([[0, 1, 0, 0, 0, 0, 1],
         [1, 0, 1, 0, 0, 0, 0],
         [0, 1, 0, 1, 0, 0, 0],
         [0, 0, 1, 0, 1, 0, 0],
         [0, 0, 0, 1, 0, 1, 0],
         [0, 0, 0, 0, 1, 0, 1],
         [1, 0, 0, 0, 0, 1, 0]]) <class 'torch.Tensor'>
```

Questions?