Traditional Methods for ML in Graphs

1. Node-Level Features

- Node Degree
- Node Centrality
  - Eigenvector centrality

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

  - Betweenness centrality

    $$c_v = \sum_{s \neq v \neq t} \frac{\text{※ shortest paths btw } s \text{ and } t \text{ that contain } v}{\text{※ shortest path btw } s \text{ and } t}$$

  - Closeness centrality

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

- Clustering Coefficient

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

  - observation: Clustering coef. counts the ※ triangles in the ego-network

- Graphlets: Rooted connected non-isomorphic subgraphs:
  - Graph Degree Vector (GDV): A count vector

of graphlets rooted at a given node.

- GDV provides a measure of node's local network topology

They can be categorized as:

- Importance-based features: Ex. predict celebrity user in social nw
  - Node degree simple
  - Different node centrality measures importance of neighboring

- Structure-based feasures: Ex. Predicting protein in protein interac. nw. functionality -protein
  - Node degree number of neighboring
  - Clustering coef. how connected neighboring nodes are
  - Graphlet count vector the occurance of diff. graphlets

Link-Level Prediction

2.1) Links missing at random

2.2) Link over time

2. Link-Level Features

- Distance-based feature

- Local Neighborhood Overlap

  - Common neighbors

$$|N(v_1) \cap N(v_2)|$$

  - Jaccard's coefficient

$$\frac{|N(v_1) \cap N(v_2)|}{|N(v_1) \cup N(v_2)|} \rightarrow \text{normalize}$$

  - Adamic-Adar index

$$\sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{\log(k_u)}$$

Limitation : metric is always zero if no common neighbors
However, the two nodes may still potentially be connected in future.

- Global Neighboring Overlap

Katz index : counts the number of paths of all length between a given pair of nodes.

compute # paths btw two nodes by Power of Adajency matrix!

Recall : $A_{uv} = 1$ if $u \in N(v)$

Let $P^{(k)}_{uv}$ = # paths of length k btw u and v

$P^{(k)} = A^k$

Katz index b/w v_1 and v_2 is calculated as sum over all path lengths

$$S_{v_1 v_2} = \sum_{l=1}^{\infty} \underbrace{\beta^l}_{0<\beta<1 \text{ discount factor}} \underbrace{A^l_{v_1 v_2}}_{\text{\# paths of length } l \text{ btw } v_1 \text{ and } v_2}$$

Katz index matrix is computed in closed-form:

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

3. Graph-Level Features

Kernel methods

- Kernel $K(G, G') \in \mathbb{R}$ measures similarity b/w data

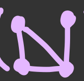
- Kernel matrix $K = (k(G,G'))_{G,G'}$ must always be PSD (positive eigenvals)
- Exists a feature representation $\phi(\cdot)$ s.t.

$$k(G,G') = \phi(G)^T \phi(G')$$

- Once the kernel is defined, off-the-shelf ML model, such as kernel SVM, can be used to make predictions

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

Key idea: Bag-of-words (BoW) for a graph

$\phi($  $) = \phi($  $)$

Bag of node degrees?

$$\begin{array}{cc} & \text{Deg} \;\; 1 \;\; 2 \;\; 3 \end{array}$$

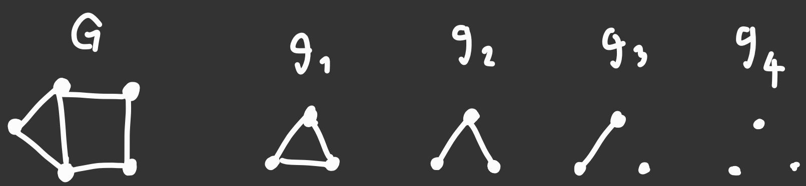
$\phi($  $) = [1, 3, 0]$

$\#$

$\phi($  $) = [0, 2, 2]$

- Graphlet Kernel (Bag-of-graphlets)

Graphlet นี้จะไม่เหมือนกับใน Node-Level แล้ว



$$f_G = (1, \quad 3, \quad 6, \quad 0)^T$$

Given two graph, G and G', graphlet kernel is

$$k(G,G') = f_G^T f_{G'}$$

Problem: if G & G' diff. size → greatly skew value

Soln: normalize

$$h_G = \frac{f_G}{\mathrm{Sum}(f_G)} \qquad k(G, G') = h_G^T h_{G'}$$

Limitation: counting graphlets is expensive!

- Weisfeiler-Lehman kernel (Bag-of-colors)

design an efficient graph feature $\phi(G)$

Color refinement

Given graph G with nodes V

- assign an initial color $c^{(0)}(v)$ to each v
- iterative refine node color by

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

- computationally efficient, closely related to GNN

Recap

$$\boxed{\text{Input Graph}} \longrightarrow \boxed{\text{Structured Features}} \longrightarrow \boxed{\text{Learning Algo.}} \longrightarrow \boxed{\text{Prediction}}$$

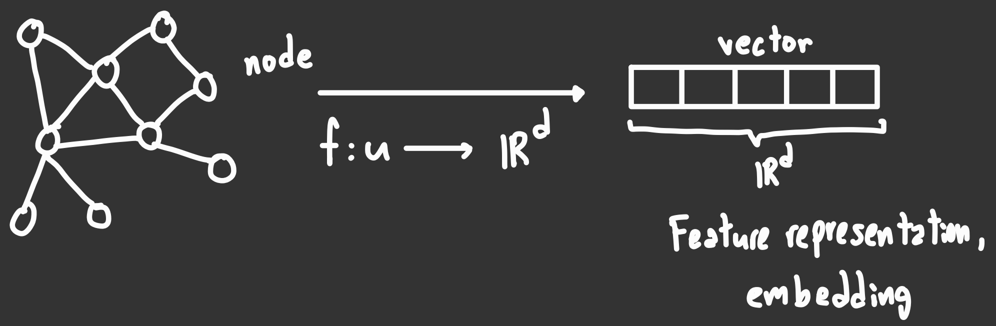
~~Feature Engineering~~
~~node/edge/graph~~
~~-level features~~
อธิบาย network

Representation Learning
automatically learn the feature

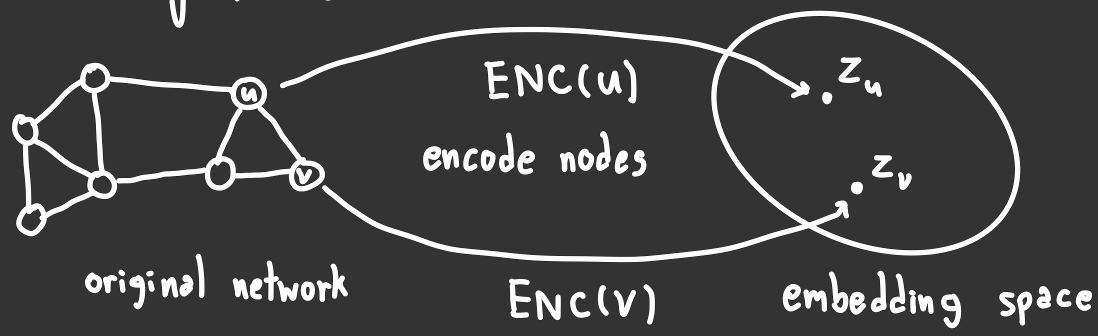
Downstream prediction task

Node Embeddings

Graph Representation Learning



Embedding Nodes



Goal: $\underset{\text{in original}}{\text{similarity}(u, v)} \approx \underset{\text{similarity of the embedding}}{z_v^T z_u}$

Learning Node Embedding

1. Encoder
2. Define node similarity function
3. Decoder, maps from embedding to the similarity score
4. Optimize the parameters of the encoder

Two Key Components

- Encoder: map each node to a low-dimensional vector

$$ENC(v) = z_v$$

- Similarity

$$\text{similarity}(u,v) \approx z_v^T z_u$$

Decoder

dot product
b/w node embedding

Random Walk Approaches for Node Embeddings

Notation:
- Vector z_u
- Probability $P(v \mid z_u)$: of visiting node v on random walk starting from node u.

Non-linear functions used to produce predicted probabilities

- Softmax function:

  Turn vector of K real values (model predictions) into K probabilities that sum to 1: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$

- Sigmoid function:

  S-shaped function that turns real values into the range of (0, 1). Written as $S(x) = \frac{1}{1+e^{-x}}$

Random-Walk Embeddings

$z_u^T z_v \approx$ probability that u and v co-occur on a random walk over the graph

Feature Learning as Optimization

- $G = (V, E)$
- Our goal is to learn a mapping $f: u \rightarrow \mathbb{R}^d$

  $f(u) = z_u$
- Log-likelihood objective:

$$\max_f \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

Random Walk Optimization

1. Run short fix-length random walks
   start from u in the graph using some random walk strategy R
2. For each node u collect $N_R(u)$, the multiset of nodes visited
   on random walks starting from u
3. Optimize embeddings according to: stochastic Gradient Descent
   Given node u, predict its neighbors $N_R(u)$

$$\max_f \sum_{u \in V} \log P(N_R(u) \mid z_u) \longrightarrow \text{Maximum likelihood objective}$$

Equivalently, $\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log \left(P(v \mid z_u) \right)$

- Intuition: Optimize embedding z_u to maximize the likelihood
  of random walk co-occurances

- Parameterize $P(v \mid z_u)$ using softmax:

$$P(v \mid z_u) = \frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)}$$

why softmax?
we want node v to be
most similar to node u
(out of all nodes n)

$\sum_i \exp(x_i) \approx \max_i \exp(x_i)$

ตอนเอาไปใส่กลับไป $O(|V|^2)$ complexity

Solution: Negative Sampling

$$\approx \log(\sigma(z_u^T z_v)) - \sum_{i=1}^{k} \log(\sigma(z_u^T z_{n_i})), \quad n_i \sim P_v$$

sigmoid function

random distribution
over nodes

Instead of normalizing w.r.t. all nodes, just normalize against k random "negative samples" n_i

- sample k negative nodes each with prob. proportional to its degree

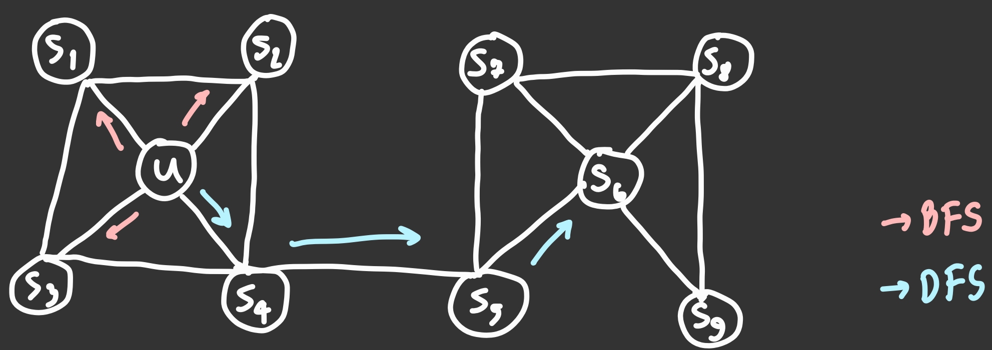
  in pratice $k = 5$-20

Overview node2vec

- Goal: Embed nodes w/ similar network neighborhoods close in the future space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- key observation: Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 2nd order random walk R to generate network neighborhood $N_R(u)$ of node u

node2vec : Biased Walk

Idea: use flexible, biased random walk that can trade off b/w local and global views of the network



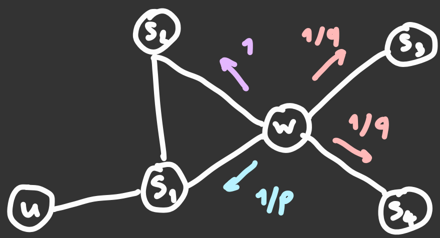
Walk of length 3 ($N_R(u)$ of size 3)

$N_{BFS}(u) = \{s_1, s_2, s_3\}$ Local microscopic view

$N_{DFS}(u) = \{s_4, s_5, s_6\}$ Global macroscopic view

Biased Random Walk

- walker came over edge (s_1, w) and is at w. Where to go next?



- BFS-like walk: Low value of p
- DFS-like walk: Low value of q

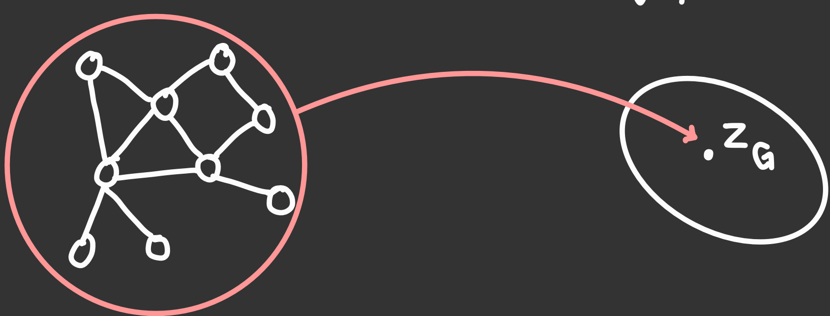
$N_R(u)$ are the nodes visited by the biased walk

node2vec algorithm

1) Compute random walk prob
2) Simulate r random walks of length l starting from each node u
3) Optimize the node2vec objective using Stochastic Gradient Descent

Embedding Entire Graphs

Want to embed subgraph or entire graph G. Graph embedding: z_G



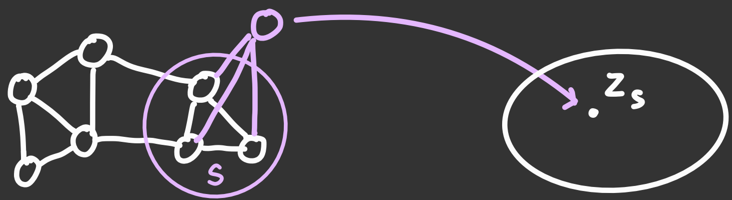
Simple idea 1:

- Run a standard graph embedding technique on the (sub)graph G

- Then just sum (or avg.) the node embedding in the G

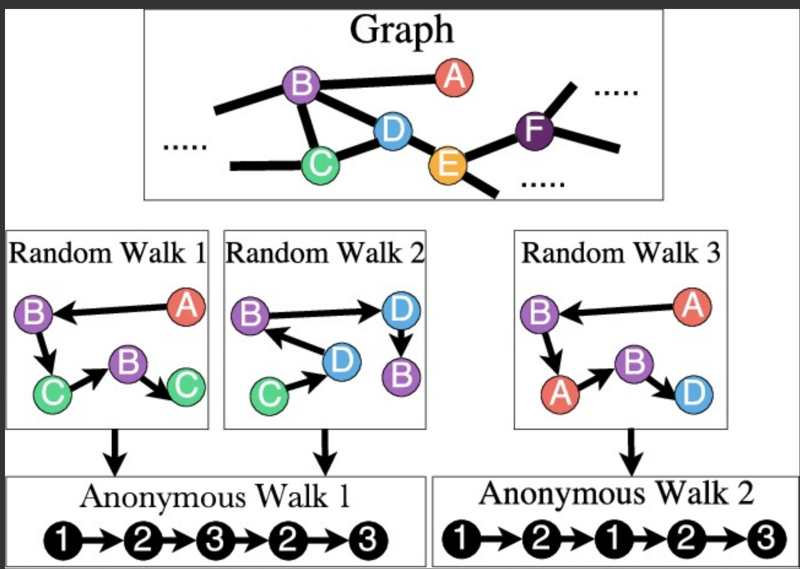
$$z_G = \sum_{v \in G} z_v$$

Idea 2: Introduce a "virtual node" to represent the G and run a standard graph embedding technique



Idea 3: Anonymous Walk Embeddings

states in anonymous walks correspond to the index of the first time we visited the node in a random walk



How many random walks m do we need?

- we want the distribution to have error of more than ε with prob. less than δ.

$$m = \left\lceil \frac{2}{\varepsilon^2} \left(\log(2^n - 2) - \log(\delta) \right) \right\rceil$$

Summary

we discussed 3 ideas to graph embeddings

- Approach 1: Embed nodes and sum/avg them
- Approach 2: Create super-node that spans the (sub)graph and then embed that node
- Approach 3: Anonymous Walk Embeddings
    - Idea 1: Sample the anon. walks and represent the graph as fraction of times each anon. walk occurs
    - Idea 2: Embed anon. walks, concatenate their embeddings to get a graph embedding

Today's summary

- Encoder-Decoder framework:
    - Encoder: embedding lookup
    - Decoder: predict score based on embedding to match node similarity
- Node Similarity measure: (biased) random walk
    - Examples: DeepWalk, Node2Vec
- Extension to Graph embedding: Node embedding aggregation and Anon. Walk Embeddings