Stanford CS 224W Notes

Yikun Chi

September 26, 2022

Contents

Chapter 1

Traditional ML Pipeline and Features

Traditionally, we use hand-designed node, link, graph level features to transform the graph into series of features, then feed into the machine learning model.

1.1 Node Level Features

Here are some important-based node features (degree, centrality) and structur-based node features (degree, clustering coefficient, graphlet count vector).

1.1.1 Node Degree

Node degree k_u just the number of edges the node u has.

1.1.2 Node Centrality

Node centrality tries to captude the node importance.

Eigenvector centrality:
We can model the centrality of a node v as the sum of the centrality of neighboring nodes. i.e.:

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

- here λ is the normalizing constant.

Turns out we can re-write it as matrix form $\lambda c_v = Ac$ where c is the centrality vector, A is the adjacency matrix, and λ is the eigenvalue. This way we have the eigenvector c_{max} correlated with the largest eigenvalue λ_{max} to be the centrality

measure.

Betweenness centrality:
A node is important if it lies on many shortest path between other nodes. So we have

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

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

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

1.1.3 Node Clustering Coefficient

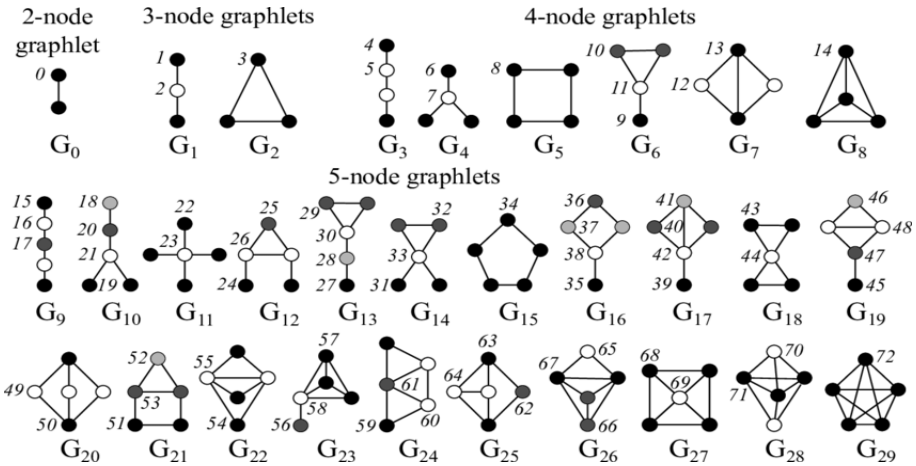
Node clustering coefficient measures how connected v's neighboring nodes are:

$$e_v = \frac{\#(\text{edges among neighboring nodes})}{\binom{k_v}{2}}$$

- k_v is the degree of the node v

1.1.4 Node Graphlets & GDV

Graphlets are small subgraphs that describe network structure around node u. Graphlets are rooted connected induced non-isomorphic subgraphs. Below shows all graphlets of a fully connected graphs with 5 nodes. Notice the numbering on the subgraphs. Since graphlets are rooted, so the nodes position matters. e.g.: in subgraph G1, a node can either be the boarder node, or the central node.



Graphlet Degree Vector (GDV) is a count vector of graphlets rooted at a given node. I.e.: at index i, the value of the vector is the number of times node u involved in graphlet configuration i.

1.2 Link Level Features

Generally, we want to predict edges in two ways: 1) assume the graph is incomplete and edges are missing at random, 2) Given $G[t_0, t_0']$ a graph defined by edges up to time t_0', output a ranked list L of edges that are not in $G[t_0, t_0']$ but would appear in time $G[t_1, t_1']$. We can do this by computing a score between each node $c(u, v)$, and use the top scores as the predicted new edge.
Link level features can be based on distance, local neighborhood overlap, and global neighborhood overlap.

1.2.1 Link Distance Based Features

Shortest-path distance between two nodes.

1.2.2 Local Neighborhood Overlap

These set of features aim to captures number of neighboring nodes shared between two nodes v_1 and v_2.
Common neighbors

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

Jaccard's coefficient

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

Adamic-Adar index

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

1.2.3 Global Neighborhood Overlap

If two nodes do not have any neighbors in common, the metric is always 0. But two nodes could still be potentially connected in the future. So we now try to consider the entire graph. If the graph is undirected, we can calculate number of paths (strictly speaking, walks) of length K be

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

- A is the adjacency matrix

- A^k_{uv} entry is the number of walk of length k between node u and v

The katz index counts the number of paths of all lengths between a given pair of nodes. So it can be calculated as

$$S_{v_1v_2} = \sum_{l=1}^{\infty} \beta^l A^l_{v_1v_2}$$
$$S = \sum_{i=1}^{\infty} \beta^i A^i = (I - \beta A)^{-1} - I = \sum_{i=0}^{\infty} \beta^i A^i$$

- β is discount factor

1.3 Graph Level Features

Graph level features aims to characterize the structure of an entire graph. Graph kernel is a function $K(G, G') \in \mathbb{R}$ that takes in two graphs, and return a number measures similarity between two graphs. So the kernel matrix K for a series of graphs must always be positive semidefinite. As a result, there exists a feature representation $\phi(\cdot)$ such that $K(G, G') = \phi(G)^T \phi(G')$. Once the kernel is defined, we can use reguar ML models such as kernel SVM.

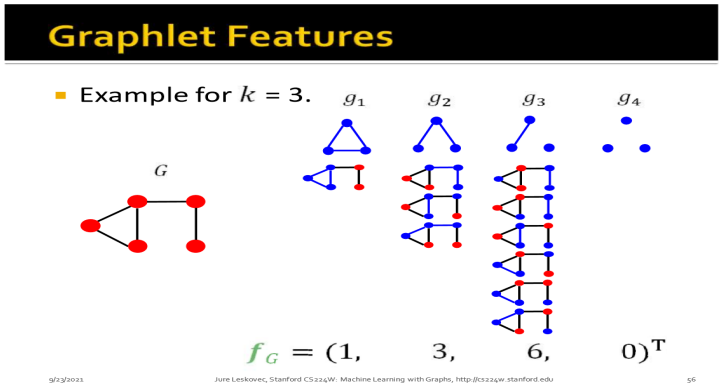
1.3.1 Graphlet Kernel

Goal: Design graph feature vector $\phi(G)$ using bag of words approach. We want to count the number of different graphlets (here it means non-rooted induced subgraph, and nodes do not need to be connected). So formally, given graph G and a graphlet list $(g_1, g_2, ..., g_{n_k})$, define the graphlet count vector $f_G \in \mathbb{R}^{n_k}$ as

$$(f_G)_i = \#(g_i \in G) \text{ for } i = 1, 2, ..., n_k$$

- Here k denotes the number of nodes in the graphlet. E.g.: when $k = 3$, there are 4 possible graphlet.

For example:



Given two graphs G and G', the graphlet kernel is computed using normalized feature vector as

$$K(G, G') = h_G^T h_{G'}$$
$$h_G = \frac{f_g}{Sum(f_G)}$$

1.3.2 Weisfeiler-Lehman Kernel

Use neighborhood structure to iteratively encirch node vocabulary. We do it through color refine algorithm.Given a graph G with a set of nodes V.

1. Assian an initial color $c^{(0)}(v)$ to each node v.
2. Iteratively refine node colors by $c^{(k+1)}(v) = HASH(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)})$ (Hash function maps different input to different color. Hash function can have collisions, but it will bring error.)
3. After K steps of color refinement, $c^{(k)}(v)$ summarizes the structure from K hop neighbors.

After color refinement, WL kernel counts number of nodes with a given color, It tracks all colors from all iterations, not just the color configuration from the

last step of color refinement. In total, the computing complexity is linear to number of edges.

Chapter 2

Node & Graph Embedding

2.1 Node Embedding Intuition

In graph representation learning, we want to skip the feature engineering, and automatically learn the features. The node embedding still should encode network structure. Assume we have a graph G, V is the vertex set, and A is the adjacency matrix (assume binary). Goal is to develop an eoncoder to encode nodes to embedding. Define a node similarity function. Decoder maps from embedding to the similarity score. Finally we can optimize the parameters of the encoder so that similarity in the embedding space (e.g.: dot product) approximates similarity in the graph.

2.2 "Shallow Encoding

The simplest encoding approach is just an embedding lookup, i.e.:

$$ENC(v) = z_v = Z \cdot v$$

- $Z \in \mathbb{R}^{d \times |v|}$ matrix, each column is a node embedding (what we learn / optimize). d is the dimensionality of the embedding layer.
- $v \in \mathbb{I}^{|v|}$ is just a indicator vector, all zeroes except a one in column indicating node v
- We use methods such as DeepWalk, node2vec to directly optimzie the embedding of each node
- Decoder is based on node similarity and the objective is to maximize $z_v^T z_u$ for node pairs (u, v) that are similar

2.3 Node Similarity with Random Walks

Quick refresher - Softmax:
Softmax function turns a vector of K real values into K probabilities that sum to 1.

$$\sigma(z)[i] = \frac{e^{z[i]}}{\sum_{j=1}^{K} e^{z[j]}}$$

Quick refresher - Sigmoid:
Sigmoid turns real values into the range of $(0, 1)$

$$S(s) = \frac{1}{1 + e^{-x}}$$

Notation:
Let z_u to be the embedding of node u, which is what we aim to find. Let $p(v|z_u)$ be the probability of visiting node v on random walks starting from node u (It also can be seen as the similarity between node u and v).

Feature Learning as Optimization:
Given $G = (V, E)$, we want to learn a mapping $f : u \to \mathbb{R}^d : f(u) = z_u$. The log-likelihood objective is

$$max_f \sum_{u \in V} logP(N_R(u)|z_u)$$

- $N_R(u)$ is the neighborhood of node u by strategy R
- So given node u, we want to learn representations that are predictive of the nodes in its random walk neighborhood $N_R(u)$

So in reality, we

1. Run short fixed-length random walks starting from each node u in the graph using some random walk strategy R
2. For each node u collect $N_R(u)$, the multset (repeated visit allowed) of nodes visited on random walks starting from u
3. Optimize embeddings according to given node u, predict its neighbors $N_R(u)$

This is equivalent to the overall loss function of

$$L = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|z_u))$$

and we can parameterize $P(v|z_u)$ with softmax:

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

But $\sum_{n \in V} e^{z_u^T z_n}$ will lead to quadratic complexity to the number of nodes, so we approximate it with negative sampling:

$$log(\frac{e^{z_u^T z_v}}{\sum_{n \in V} e^{z_u^T z_n}}) \approx log(\sigma(z_u^T z_v)) - \sum_{i=1}^{K} \log(\sigma(z_u^T z_{n_i}))$$

- σ is the sigmoid function
- $n_i \sim P_V$ is random distribution over nodes. We want to sample k negative node n_i each with probability proportional to its degree.
- higher K gives more robust estimates, but also corresponds to higher bias on negative events. In practice, let $k = 5 - 20$
- Normally people just sample K from any nodes, but ideally it should be nodes not on the walk

We then solve z_u with stochastic gradient descent (SGD)

- Initialize z_u at some randomized value for all nodes u
- iterate until convergence, at every step:
  - sample a node u, for all v, calculate derivative $\frac{\partial}{\partial z_v} L(u)$
  - For all v, update $z_v \leftarrow z_v - \eta \frac{\partial}{\partial z_v} L(u)$

2.3.1 Random Walk Strategy: node2vec

The simplest random walk strategy would just be unbiased random walks (i.e.: DeepWalk). A better approach biased second order random walk R. We have two parameters:

- Return parameter p: return back to the previous node
- In-out parameter q: Moving outwards (DFS) vs. inwards (BFS). Intuitively, q is the "ratio" of BFS vs. DFS

Say we start at node u, and after some step we are at node v, we travel to the following types of nodes with unnormalized probabilities:

- nodes that are closer to u with $\frac{1}{p}$
- nodes with same distance from u (1 link away) with 1
- nodes that are further away from u with $\frac{1}{q}$

So the overall node2vec algorithm is

1. Compute random walk probabilities
2. Simulate r random walks of length l starting from each node u
3. Optimize the node2vec objective using SGD

2.4 Node Embedding to Graph Embedding

Naive approach: sum up all node embedding. New approach: introduce a "virtual node" to represent the subgraph / graph, and run a standard graph embedding technique.

2.5 Limitation of Node Embeddings

- We cannot obtain embeddings for nodes not in the training set. If a new node is added, we have to recalculate all embedding with DeepWalk and node2vec
- Cannot capture structural similarity. (e.g.: same node in rooted graphlet)
- Cannot utilize node, edge, and graph features.

Chapter 3

PageRank and Matrix Formulation of Graph

3.1 PageRank

Rank
We can define the rank r_j for node j as

$$r_j = \sum_{i \to j} \frac{r_j}{d_j}$$

So essentially we are splitting the node i rank by its degree, and flow it to node j.

Stochastic Adjacency Matrix M
Let d_i to be the outdegree of node i, if $i \to j$, then $M_{ji} = \frac{1}{d_i}$. So each columns of M sums to 1. Let r be the rank vector, hence r_i is the importance score of page i. Then, we can write the entire flow of the graph as

$$r = M \cdot r$$

or

$$\forall j, r_j = \sum_{i \to j} \frac{r_i}{d_i}$$

PageRank to Random Walk and Eigenvector
Let vector $p(t)$ denotes the probability of a random surfer is a certain page (e.g.: $p(t)[i]$ indicates the probability of the random surfer at page i at time t. Suppose the random walk reaches a stationary state, i.e.: $p(t+1) = M \cdot p(t)$, then $p(t)$ is the stationary distribution of a random walk. But note this is same equation as the rank vector. So r is a stationary distribution for the random

walk.

The rank vector r is an eigenvector of the stochastic adjacency matrix M with eigenvalue of 1. Given any starting position vector u, the limit $M(M(...M(Mu)))$ is the long-term distribution of the surfer, which is r. So r is the principal eigenvector of M with eigenvalue of 1, and can be solved via power iteration.

In summary, we can solve r as $M(M(...M(Mu)))$. Alternatively, we can just update node by node with $r_j^{(t+1)} = \sum_{i \to j} \frac{r_i^{(t)}}{d_i}$ with random initiation (such as uniform initiation).

Accounting for Dead-end and Spider Traps
Two problems:

- Dead-end (no out-links). Such pages cause importance to leak out. Cause the in-flow will not flow to other place.
- Spider traps (all out-links are within the group). Eventually spider traps will obsort all importance

Solutions:

- For dead-end, we want to follow random teleport links with total probability of 1 from the dead-ends (adjust matrix accordingly).
- For spider traps, at each time step, the random surfer has probability of β to follow a link at random, and $1-\beta$ jump to a random page. Common value for β is 0.8 and 0.9.

Overall, we have

$$r_j = (\sum_{i \to j} \beta \frac{r_i}{d_i}) + (1-\beta)\frac{1}{N}$$

or

$$G = \beta M + (1-\beta)[\frac{1}{N}]_{N \times N}$$

We can recursively solve the problem of $r = G \cdot r$ using the same process. Note this assumes that the graph has no dead-end. We can preprocess the graph by altering the dead-end column in adjacency matrix M to be uniformally teleport.

- In reality, we never materialize matrix G due to space complexity. We want to work with sparse matrix M and the implicit teleport matrix.
- Note the dead-end pre-processing is just a one-time process

3.2 Personalized PageRank

We want to use PageRank to measure proximity of the nodes. Given a set of query nodes, we simulate a random walk. As we walk, we record the visit counts of each neighbor. But with probability α, we restart the walk at one of the query nodes. The highest visit count nodes has the highest proximity to nodes. Note this teleportation can be calculated in Matrix form by setting the teleportation matrix. In page rank, the teleporation vector (one column in the teleporation Matrix) is $[\frac{1}{N}]$. In Personalized PageRank, we restrict teleport to only a set of nodes. But be careful to set the query nodes to be in a Spider Traps.

3.3 Random Walk with Restarts

For random walk with restart, it is essentially same as personalized PageRank, but only restart at a specific node instead of a set of query nodes. But be careful to set the query nodes to be in a Spider Traps.

3.4 Matrix Factorization and Node Embeddings

Recall in node embedding, we have an embedding matrix Z. Each column of the Z is an embedded node, and we want to maximize $z_v^T z_u$ for nodes u, v that are similar.

Edge as similarity
If we use having an edge as similarity, then we have $z_v^T z_u = A_{u,v}$, hence $Z^T Z = A$. In reality, we can't achieve exact factorization. So we want to minimize $\left\|A - Z^T Z\right\|$ (here we use L2 norm, and previously we used softmax)

DeepWalk similarity
DeepWalk define node similarity based on uniform random walks of the fixed length. It is equivalent to matrix factorization of the following matrix

$$\log\left(vol(G)(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r)D^{-1}\right) - \log b$$

- $vol(G)$ is $2 \times$ # of edges
- $T = |N_R(u)|$, or the conext window size / length of random walk
- D is the diagonal matrix D with entries to be the nodes degree
- b is the number of negative samples.

Node2vec can also be formulated as matrix factorization, but more complicated.

Chapter 4

Message Passing and Node Classification

Primer
Given a network with labels on some nodes, how do we assign labels to all other nodes in the network. We assume similar nodes are connected (i.e.: homophily) in the network.

Homophily
Individual characteristics of the nodes leads to social interaction. So because nodes are similar, they are connected.

Influence
Due to connection, nodes are influencing each other. So because nodes are connected, they are similar.

4.1 Relational Classification

We want to propagate node labels across the network. For labeled node v, initialize label Y_v with ground-truth label $Y*_v$. For unlabeled nodes, initialize $Y_v = 0.5$, then update all nodes in a random order until convergence or until maximum number of iterations is reached.
The update step for each node v and label c is :

$$P(Y_v = c) = \frac{1}{\sum_{(v,u)\in E} A_{v,u}} \sum_{(u,v)\in E} A_{v,u} P(Y_u = c)$$

- If edges have strength / weight information, $A_{v,u}$ can be the edge weight between v and u. Ohterwise, use adjaency matrix.

- Model convergence is not guaranteed
- Model does not use node feature information
- Note we always uses the most upated probability. So if a node's probability is updated, then when it comes to updating the node's neighbor, we use the updated probability of the original node.

4.2 Iterative Classification

The main idea is to classify node v based on its attributes f_v as well as labels z_v of neighbors set N_v. The approach is to train two classifies:

- $\phi_1(f_v)$ Predict node label based on node feature vector f_v. We use this to initialize our iteration.
- $\phi_2(f_v, z_v)$ Predict label based on node feature vector f_v and summary z_v of labels of v's neighbors

The summary z_v of labels of v's neighbors N_v is a vector that capture labels around node v, it can include

- Histogram of the number / fraction of each label in N_v
- Most common label in N_v
- Number of different labels in N_v
- Features from neighbor nodes

So the overall architecture is

- Phase 1: Classify based on node attributes alone
  - On a training set, we train two classifies:
  - $\phi_1(f_v)$ to predict Y_v based on f_v
  - $\phi_2(f_v, z_v)$ to predict based on f_v and z_v (Note here we do not use ϕ_1 result to train for ϕ_2. ϕ_2 is trained using only truth)
- Phase 2: Iterate till convergence
  - On test set, set labels Y_v based on the classifier ϕ_1, compute z_v, and predict the labels with ϕ_2
  - Repeat for each node v:
    - Update z_v based on Y_u for all $u \in N_v$
    - Update Y_v based on the new z_v using ϕ_2 if the label has no ground truth.
  - iterate until class labels stabilize or max number of iterations is reached

Note we do not re-train the classifier.

4.3 Correct and Smooth

the base model can make different degrees of error at different location. So we need to correct the bias, and smoothen it over the graph. Correct and Smooth follows the three-step procedure:

1. Train base predictor
2. Use the base predictor to predict soft labels of all nodes
3. Post-process the predictions using graph structure to obtain the final predictions of all nodes.

Base Predictor
We train a base predictor to predict label. It could be a linear model or MLP over node features. We then use the base predictor to apply the soft labels for all the nodes. E.g.: the probability vector for a node v could be $[0.95, 0.05]$ for a binary classification.

Correct Step
Compute the training errors of node as ground-truth label minus soft label. Defined as 0 for unlabeled nodes. E.g.: if a node's ground truth is $[1, 0]$ and the soft label is $[0.95, 0.05]$, the error is $[0.05, -0.05]$.

We then diffuse training error $E^{(0)}$ along the edges. Let A be the adjacency matrix, $\tilde{A}$ be the diffusion matrix, then we have

$$E^{(t+1)} \leftarrow (1-\alpha) * E^{(t)} + \alpha * \tilde{A} E^{(t)}$$

- Set adjacency matrix A with self loop. i.e.: set $A_{ii} = 1$
- let $D = Diag(d_1, ..., d_N$ be the degree matrix
- The diffusion matrix $\tilde{A}$ could be $D^{-1/2} A D^{-1/2}$

After several step of diffusion, we add a scaled diffused training errors into the predicted soft label for all nodes. So for each node, the new soft label is $old_label + s * error$. s can be equal to 2, or tuned as a hyper-parameter.

Smooth Step
The input to smooth steps are the post-correction soft label for unlabeled nodes, and ground truth hard label for labeled nodes. We then diffuse label $Z^{(0)}$ along the graph structure with

$$Z^{(t+1)} \leftarrow (1-\alpha) * Z^{(t)} + \alpha * \tilde{A} Z^{(t)}$$

After smooth step, we can predict based on normalized soft label. Generally, we do about 10 iteration of correct step and smooth step.

Chapter 5

Graphical Neural Network

5.1 Primer

Recall for node embeddings, we want to map node to a d dimensional embeddings such that similarity(decoder) of those two embeddings are close for nodes close to each other. Now we use deep graph encoders. This can solve

- node classification
- link prediction
- community detection
- network similarity

5.2 Deep Learning for Graphs Requirement

Setup
Assume we have a graph G:

- V is the vertex set
- A is the adjacency matrix
- $X \in \mathbb{R}^{m \times |V|}$ is a amtrix of node feature
- v is a node in V. $N(v)$ is the neighbors.

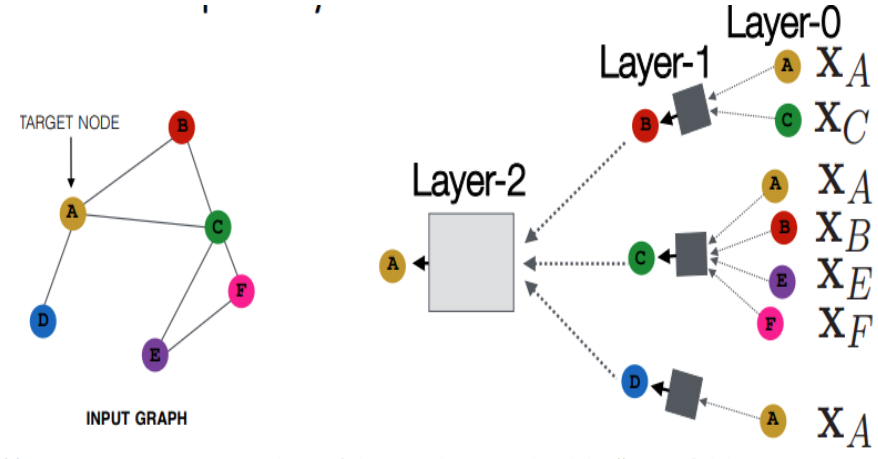
Notice for a given graph, we have many ordering plans for the nodes. So our algorithm has to be invariance / equivariance.

Permutation Invariance and Equivariance
Let f be a function that maps a graph $G = (A, X)$ to a vector $\mathbb{R}^d$. Then we say f is permutation invariant function if we can shuffle the order of the columns of A and X (i.e.: different order plan for nodes).

Let f be a function that maps a graph $G = (A, X)$ to $\mathbb{R}^{m \times d}$ where m is the number of nodes. The output each row is the embedding of a node. We say f is equivariance if we can shuffle the order of the input, and the embedding output for the same node is the same.

5.3 Graph Convolutional Networks - Basic Approach

Idea: We let a node's neighborhood defines a computation graph. So below is an example of a 2 layer GNN setup for a specific node. Each node will have a computation graph.



So we have a two step process at each layer: 1) average messages from neighbors and 2) apply neural network.

$$h_v^0 = x_v \qquad \text{(Initial 0-th layer embeddings are equal to node features)}$$

$$h_v^{k+1} = \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(V)|} + B_k h_v^{(k)}) \qquad \forall k \in \{0, ..., K-1\}$$

$$z_v = h_v^K$$

- $\sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(V)|}$ is averaging the embeddings of neighbors. (This is order invariant)
- $B_k h_v^{(k)}$ is just the embedding of the same node in k layer.

- σ is an activation function
- W_k is shared across all nodes. W is only different at different layer.
- We can also think of the process as $\sum_{u \in N(v)} W_k \frac{h_u^{(k)}}{|N(V)|}$. So each neighbor's message for layer $k+1$ is computed as $W_k \frac{h_u^{(k)}}{|N(V)|}$, and the aggregation process is a sum.

5.4 Matrix Formulation of the simple GNN with average aggregation

Many aggregations can be performed efficiently by sparse matrix operations.

Let $H^{(k)} = [h_1, ..., h_{|v|}^{(k)}]^T$

$\implies \sum_{u \in N_v} h_u^{(k)} = A_v H^{(k)}$ (A_v is the row for node v in adjacency matrix)

Let D be a diagonal matrix and $D_{v,v} = Deg(v) = |N(v)|$

$\implies H^{(k+1)} = \sigma(D^{-1} A H^{(k)} W_k^T + H^{(k)} B_k^T)$

5.5 Training Simple GNN

The mode parameters are W_i and B_i. I will have one W and one B per layer. So we just need to decide a loss function.
For supervised learning, the node embedding z_v is a function of input graph. We can try to minimize the loss $L(y, f(z_v)$
In an unsupervised setting, we can use the graph structure as the supervision. Hence $L = \sum_{z_w, z_v} CE(y_{u,v}, DEC(z_u, z_v))$. The CE is cross entropy. DEC is decoder such as inner product. We can set $y_{u,v} = 1$ when node u and v are similar, and node similarity can be anything from previous lecture such as random walks.

5.6 GNN Component Summary

In a general GNN framework, we have 4 components:

- Messaging + Aggregation
- Layer Connectivity
- Graph Augmentation (input graph $\neq$ computational graph): graph feature augmentation, graph structure augmentation
- Learning objective: supervised vs. unsupversed, node/edge/graph level objectives

5.7 Single Layer of GNN

Single layer of GNN has two component:

Message Computation
Each node will create a message, which will be sent to other nodes later:

$$m_u^{(l)} = MSG^l(h_u^{(l-1)}$$

Some example could be a linear layer $m_u^{(l)} = W^{(l)} h_u^{(l-1)}$.
In addition, we want to compute a different message for the node v itself, that will be used for self node's message at next layer:

$$m_v^{(l)} = MSG(h_v^{(l-1)}) = B^{(l)} h_v^{(l-1)} \qquad \text{(example of linear embedding)}$$

Aggregation
Node v will aggregate the message from node v's neighbors as well as the message from itself from previous layer.

$$h_v^{(l)} = AGG^{(l)}(\{m_u^{(l)}, u \in N(v)\}, m_v^{(l)})$$

The aggregation could be using sum, mean, max and etc. The way to incorporate self message could be through concatenation, or aggregation.

5.7.1 GraphSAGE Layer

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot CONCAT\left(h_v^{(l-1)}, AGG(\{h_u^{(l-1)}, \forall u \in N(v)\})\right)\right)$$

- Message is computed within the $AGG(\cdot)$
- First we aggregate from node neighbors, $AGG(\{h_u^{(l-1)}, \forall u \in N(v)\})$
- then we further aggregate over the node itself through concatenation

Different kind of aggregator:

- Mean: Take a weighted average of neighbors $AGG = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}$
- Pool: Transform neighbor vectors (e.g.: Multilayer Perceptron) and apply symmetric vector function Mean or Max: $AGG = Mean\left(\{MLP(h_u^{(l-1)}), \forall u \in N(v)\}\right)$
- LSTM: Apply LSTM to **reshuffled** of neighbors (so model doesn't learn the sequence) $AGG = LSTM([h_u^{(l-1)}, \forall u \in N(v)]$

L2 Normalization:

- We can apply l2 normalization to $h_v^{(l)}$ at every layer

5.7.2 Graph Attention Networks Layer

We can have an weight factor for each edge a_{vu}

$$h_v^{(l)} = \sigma(\sum_{u \in N(v)} a_{vu} W^{(l)} h_u^{(l-1)})$$

- In GCN / GraphSage, $a_{vu} = \frac{1}{|N(v)|}$ is the weighting factor of node u' message to node v. All neighbors are equivalent important.
- Ideally we want to let different neighbor have different attention

Overall, we want to let the attention a_{vu} be computed as a byproduct of an attention mechanism function α

First we compute attention coefficient based on messages

$e_{vu} = \alpha(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)})$

Normalize e_{vu} into the final attention weight with softmax

$$a_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

The approach for attention mechanism function α can be a simple single layer neural network, e.g.: (The parameters of α are trained jointly with other parameters in end-to-end function.)

$$\alpha(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)}) = Linear(Concat(W^{(l)} h_u^{(l-1)}, W^{(l)} h_v^{(l-1)}))$$

Multi-head attention: Stablizes the learning process of attention mechanism. We create multiple attention scores, each generate a new $h_u^{(l1)}$, and then we can have an aggregation function to aggregate all different version of the message, such as sum, concatenation, or average.

5.8 Layer Connectivity

Traditionally, we can construct Graph Neural Network just by stacking GNN layer sequentially. The input is the initial raw node feature x_v and the output is the node embeddings h_v^L after L GNN layers.

Over-smoothing
Receptive field is a set of nodes that determine the embeddings of a node of interest. So for a two layer GNN, the receptive field is a node's 2 hop neighbors. As we add layer to GNN, there will be a lot of overlap in terms of receptive field, and thus eventually nodes embedding will converge to the same value.

Solution 1: We can make shallow GNN more expressive by making aggregation / transformation become a deep neural network.

Solution 2: Add layers that do not pass messages. A GNN layer does not necessarily only contain GNN layer, we can add MLP layers (applied to each node) that does pre-process and post-process.

Solution 3: Add skip connection in GNNs. So normally a standard GCN layer is $h_v^{(l)} = \sigma(\sum_{u \in N(v)} W^{(l)} \frac{h_u^{(l-1)}}{|N(v)|})$, but now we have skipped connection: $h_v^{(l)} = \sigma(\sum_{u \in N(v)} W^{(l)} \frac{h_u^{(l-1)}}{|N(v)|} + h_v^{(l-1)})$. Alternatively, we can also let the final layer directly aggregates from all the node embeddings in the previous layers.

5.9 Graph Augmentation

Our raw input graph doesn't have to be same as our computational graph.

5.9.1 Feature Augmentation

If the input graph does not have node feature

- Assign a constant to each node: medium expressive power, inductive, and low computational cost.
- Assign unique ID to each node with one-hot encoding: highly expressive, not inductive, high computational cost due to $O(|V|)$ dimensitonal feature.

We can also strategically add certain structural features to node features, such as cycle length, PageRank, Centrality, Node degree, clustering coefficient.

5.9.2 Graph Structure Augmentation

5.9.2.1 Adding virtual nodes / edges

To combat super sparse matrix, we can add virtual edges,

- Connect 2-hop neighbors via virtual edges (so instead of use GNN computation, we use $A + A^2$
- Just use the 2-hop virtual edges (This is especially useful for bipartie graphs.)

We can also add virtual nodes that connect to all the nodes in the graph. Those nodes will improves message passing in sparse graphs

5.9.2.2 Sample neighbors when do message passing

If the graph is too dense, we can randomly sample a node's neighborhood for message passing. We can even use random walk to sample neighbors.

5.10 Learning Objective

5.10.1 Node-Level Prediction

After GNN computation, we have d-dimensional node embeddings : $h_v^{(L)} \in \mathbb{R}^d$. For classification, We can just wrap the head node with another matrix W^H to map the embeddings to linear output and take softmax to compute loss.

5.10.2 Edge-Level Prediction

We can make edge prediction by taking two head node. $\hat{y}_{u,v} = Head(h_u^{(L)}, h_v^{(L)})$. For design of the head function that aggregates, we can do

- concatenation and then linear layer (not recommended)
- dot product for one-way prediction
- Multi-headed trainable matrix $W^{(1)}, ..., W^{(k)}$, $\hat{y}_{uv}^{(i)} = (h_u^{(l)})^T W^{(i)} h_v^{(l)}$ and then concatenate all the $\hat{y}^{(i)}$

5.10.3 Graph / Subgraph Level Prediction

For graph level prediction of a smaller graph, we can think of it as aggregation over all embeddings, such as averaging, summing, maxing. For large graph, we can do hierarchically pooling. For example, we can use $RELU(Sum(\cdot))$. So we can divide nodes into different batches in hierarchically structure, run $RELU(Sum(\cdot))$ on each batch, and then run $RELU(Sum(\cdot))$ on the result, and so on. DiffPool has two GNN, one to generate aggregate node, and one to create series of hierarchical cluster assignment. We can also do this many times and have two GNN for each layer of the aggregation.

5.10.4 Loss function

In supervised learning, we can use ground truth. In unsupervised learning, the signals come from graph themselves such as clustering coefficient.
For classification loss, we can use Cross Entropy loss in a K-way prediction for the ith data point

$$CE(\hat{y}^{(i)}, y^{(i)}) = -\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$

For k-way regression, we have K-way MSE

5.11 Graph Training

5.11.1 Transductive vs. Inductive Setting

5.11.1.1 Transductive Setting

The dataset consists of one graph. The input graph can be observed in all the dataset splits (training, validation, and test set). We only split the labels. Applicable to node / edge prediction tasks.

5.11.1.2 Inductive Setting

We break the edges between splits to get multiple graphs or the dataset consists of multiple graphs. Each split can only observe the graphs within the split. Applicable to node / edge / graph tasks.

5.11.2 Node Classification

Transductive Node Classification
AT training time, we compute embeddings using entire graph, and train using only the training nodes(Compute message passing using the graph, but the loss function only involves training nodes). At validation time, we compute embeddings using the entire graph, and evaluate on the validation nodes.

Inductive Node classification
At training time, we compute the embeddings and train using the training graph. At validation time, we compute and evaluate using the validation graph. (Transfer the learned parameter across).

5.11.3 Graph Classification

Graph level prediction is automatically inductive setting.

5.11.4 Link Prediction

For link prediction, we need to create the labels and dataset splits on our own. We need to hide some edges from the GNN and let the GNN predict if the edges exist.
Step 1: Assign 2 types of edges in the original graph:

- Message edges: used for GNN message passing
- Supervision edges: use for computing objectives

After step 1, only message edges will reamin in the graph. In step 2, we need to split edges into train / validation / test

- Transductive link prediction: At training time, use training message edges to predict training supervision edges. At validation time, using training message edges + training supervision edges to predict validation edges. At test time, use training message edges, training supervision edges, and validation edges to predict test edges.
- Inductive link prediction split: Given multiple graphs or split the graph into multiple graphs. Each graph will still have message edges and supervision edges.

5.12 Graph Isomorphism Network

Most expressive GNN should map subtress to embedding in an injective fashion. Subtress of the same depth can be recursively characterized from the leaf nodes to the root nodes. So the most expressive GNN would use an injective neighbor aggregation.
Theory: Any injective multi-set function can be expressed as :

$$\Phi(\sum_{x \in S} f(x))$$

- Φ and f are some non-linear function

Universal Approximation Theorem: 1-hidden-layer MLP with sufficiently-large hidden dimensionality and appropriate non-linearity $\sigma(\cdot)$ can approximate any continuous function to an arbitrary accuracy.

Now we have arriaved at GIN, MLP + element-wise sum + MLP:

$$MLP_{\Phi}(\sum_{x \in S} MLP_f(x))$$

5.12.1 GIN to WL Graph Kernel

WL Kernel:
Given A graph G with a set of node V

- Assign an initial color $c^{(0)}(v)$ to each node v
- Iteratively refine node colors by $c^{(k+1)}(v) = HASH\left(c^{(k)}(v), \{c^{(k)}(u)_{u \in N(v)}\}\right)$

GIN uses a neural network to model the injective hashing function (here ϵ is a learnable scalar to differentiate self node and neighbor nodes.

$$MLP_{\Phi}\left((1+\epsilon)MLP_f(c^{(k)}(v)) + \sum_{u \in N(v)} MLP_f(c^{(k)}(u))\right)$$

Chapter 6

Heterogeneous Graphs

6.1 Heterogeneous Graph Definition

A heterogeneous graph is defined as

$$G = (V, E, R, T)$$

- Nodes with node type $v_i \in V$
- Edges with relation type $(v_i, r, v_j) \in E$
- Node Type $T(v_i)$
- Relation type $r \in R$

6.2 Relational GCN

Extending to heterogeneous graph from the GCN by using different neural network weights for each relation type

$$h_v^{(l+1)} = \sigma\left(\left(\sum_{r \in R} \sum_{u \in N(v)^r} \frac{1}{c_{v,r}} W_r^{(l)} h_u^{(l)}\right) + W_0^{(l)} h_v^{(l)}\right)$$

- $c_{v,r}$ is the node degree of the given relationship

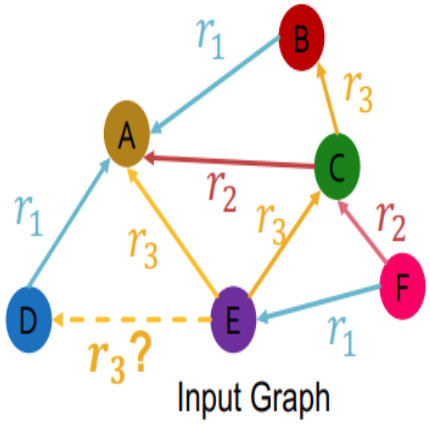
But this way we have rapids growth in number of parameters w.r.t to number of relations. So overfitting become an issue. So we use the following two methods to regularize the weights $w_r^{(l)}$:

- Use block diagonal matrices for neural network
- Basis/Dictionary learning: Represent the matrix of each relation as a linear combination of basis transformations $W_r = \sum_{b=1}^{B} a_{rb} \cdot V_b$ where V_b is shared across all relations. So now each relation only needs to learn $\{a_{rb}\}$ which is B scalars and V_b basis matrices which is B matrices.

6.2.1 Link Prediction with GCN

For link prediction, now we need one prediction head for each type of relations since there could be multiple edge type. So assume (E, r_3, A) is training supervision edge and all the other edges are training message edges. We need to

- Take the final layer of E and A: $h_E^{(l)}$ and $h_A^{(l)} \in \mathbb{R}^d$ and relation-specific scoring function (one function per relationship). E.g.: $f_{r_1} = h_E^T W_{r_1} h_A$
- Here W_{r_1} is a new matrix specifically for getting the relation specific scoring.



In practice, we

1. Use RGCN to score the training supervision edge (E, r_3, A)
2. Create negative edge by pertubing the supervision edges. e.g.: (E, r_3, B) if E and B are not connected in real graph. Note that the negative edges should not belong to training message edges or training supervision edges.
3. Use GNN model to score negative edge
4. Optimize a standard cross entropy loss (Maximize the score of training supervision edge and minimize the score of negative edge).

At evaluation time, let's say our target validation edge is (E, r_3, D)

1. Calculate the score of (E, r_3, D)
2. Calculate the score of all the negative edges $(E, r_3, v)|v \in (B, F)$ since (E, r_3, A) and (E, r_3, C) belong to training message edges and training supervision edges.

3. Obtain the rank of (E, r_3, D) We use rank because the result will be highly imbalanced class due to negative sampling

4. Calculate metrics: 1)Hits at k : $\mathbb{I}(RK \leq k)$, higher is better. 2): Reciprocal Rank $\frac{1}{RK}$, higher is better

6.3 Knowledge Graph: Predict Missing Tails

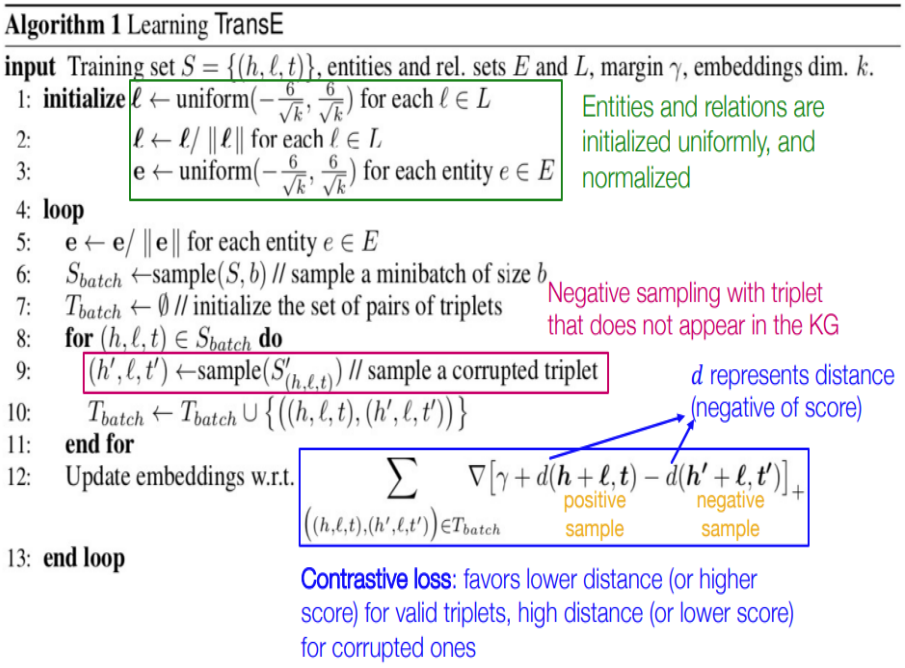
Given an enormous knowledge graph, and a head and relation, we want to predict missing tails. Edges in KG are represented as triples (h, r, t). The key idea is to model entities and relations in the embedding space $\mathbb{R}^d$. So we want to make the embedding of (h, r) to be close to the embedding of t

6.3.1 Relation Type

Relations can be Symmetric, Inverse, Composition (Transitive) and 1 to N

6.3.2 TranSE

Intuition: For a triple $(h, r, t), h, r, t \in \mathbb{R}^d$, we want $h + r \approx t$ if the given fact is true. The scoring function is $f_r(h, t) = -\|h + r - t\|$. It cannot model symmetric and 1-to-N relations



6.3.3 TranR

Model entities as vectors in the entity space $\mathbb{R}^d$ and model each relation as vector in relation space $r \in \mathbb{R}^k$ with $M_r \in \mathbb{R}^{k \times d}$ as the projection matrix that can transform entity to relation space. So the scoring function is $f_r(h,t) = -\|M_r h + r - M_r t\|$. It can model symmetric, antisymmetric, 1-to-N relation and inverse relationships. But it can not model transitive relations.

6.3.4 Bilinear Modeling

Entities and relations using vector in $\mathbb{R}^k$. The scoring function $f_r(h,t) =< h, r, t >= \sum_i h_i \cdot r_i \cdot t_i$. It can model 1-to-N relations, symmetric, but not anti-symmetric, transitive and inverse.

6.3.5 ComplEX

Based on the same scoring function as bilinear modeling, but emebds entities and relations in complex vector space $\mathbb{C}^k$. So the scoring function is $f_r(h,t) = Re(\sum_i h_i \cdot r_i \cdot \bar{t_i})$. $\bar{t_i}$ is the complex conjugate. It can model antisymmetric, symmetric, inverse, 1-to-N but can not model transitive relation.

6.4 Knowledge Graph: Multi-hop queries

Knowledge graph completion problems can be reformulated as answering one-hop queries. i.e.: is link (h,r,t) exists in the KG is equivalent to is t an answer to query (h,r). In general, we can formulate an $n-$ hop path query as

$$q = (v_a, (r_1, ..., r_n))$$

- v_a is an "anchor" entity
- The answer in graph G is denoted by $[[q]]_G$

The goal is to answer path-based queries over an incomplete knowledge graph. So we are implicitly doing graph completion.

6.4.1 Query2Box Intuition

Map queries into embedding space and reason in the embedding space. Query2Box embed query into a hyper-rectangle (box) in the Euclidean space, the answer nodes are enclosed in the box.

TransE translate h to r with score function $f_r(h,t) = -\|h + r - t\|$. Another way to look at is that the query embedding $q = h + r$, and we want query embedding q close to the answer embedding t. i.e.: $f_q(t) = -\|q - t\|$. We can easily extend TransE to handle compositional relations, i.e.: multi-hop queries.

We can embed queries with hyper-rectangles with a center and offset: $q = (Center(q), Offset(q))$. In this formulation, the intersection of boxes is well-defined (another box). So we can handle "and" logic.

6.4.2 Query2Box Embedding Details

For Query2Box, we need to figure out the following embedding / operations:

- Entity embeddings (zero-volume box)
- Relation embeddings (take a box and produces a new box)
- Intersection operator f (take multiple boxes and output a box).

6.4.2.1 Relation Embeddings with Projection Operator

We can model relation embeddings with projection operator. It takes the current box as input and project and expand the box with a new center and offset.

6.4.2.2 Intersection Operator

Geometric intersection operator j can be defined as

$$Cen(q_{inter}) = \sum_i w_i \odot Cen(q_i)$$
$$w_i = \frac{\exp(f_{cen}(Cen(q_i)))}{\sum_j \exp(f_cen(Cen(q_j)))}$$
$$Off(q_{inter}) = min(off(q_1), ..., off(q_n)) \odot \sigma(f_{off}(off(q_1), ..., off(q_n)))$$

Where w_i is calculated by a neural network f_{cen}. The offset is also calculated with neural network with sigmoid function

6.4.2.3 Entity-to-Box Distance

Given a query box q and entity embedding box v, we can define the distance (L1 Distance) as

$$d_{box}(q, v) = d_{out}(q, v) + \alpha * d_{in}(q, v) \qquad 0 < \alpha < 1$$

So we let the distance in the box to be down weighted. Finally we let the score $f_q(v) = -d_{box}(q, v)$

6.4.2.4 Dealing with Union Operation

Since the union of boxes is not box. We can not establish a simple union operator. But we can take all the unions out and only do union at the last step (outside the embedding space).

In this way, the distance between query and an entity when dealing with "or" logic can be defined as

$$q = q_1 \vee q_2 \vee ... \vee q_m \qquad d_{box}(q, v) = min(d_{box}(q_1, v), ..., d_{box}(q_m, v))$$

Here each q_i is a multi-hop query that does not contain "or" logic.

6.4.3 Training Query2Box

Given a query embedding q, we want to maximzie the score $f_q(v)$ for answers $v \in [[q]]$ and minimize the score $f_q(v')$ for negative answers $v' \notin [[q]]$. The trainable parameters are :

- Entity embeddings $d|v|$ (d dimension vector times vertiecs)
- relation embeddings $2d|R|$ (one set for move the center, one set for scale the box)
- intersection operator

So the overall training summary is :

- Randomly sample a query q from the training graph G_{train}, answer $v \in [[q]]_{G_{train}}$ and negative sample $v' \notin [[q]]_{G_{train}}$. Negative sample has to have the same entity type as the positive answer.
- Embed the query q
- Calculate the score $f_q(v)$ and $f_q(v')$
- Optimize the loss l to maximize $f_q(v)$ while minimize $f_q(v')$. i.e.: $l = -\log \sigma(f_q(v)) - \log(1 - \sigma(f_q(v')))$

For the queries, we can sample them from a query template. We normally start with the answer and query template to find the anchor nodes. (Check lecture slides for detail).

Chapter 7

Subgraph Mining

7.1 Definition of Subgraphs

7.1.1 Node-induced subgraph

Take subset of the nodes and all edges induced by the nodes: i.e.: $G' = (V', E')$ is a node induced subgraph iff

$$V' \subseteq V$$
$$E' = \{(u, v) \in E | u, v \in V'\}$$

This is also normally called node induced subgraph

7.1.2 Edge-induced subgraph

$G' = (V', E')$ is an edge induced subgraph iff

$$E' \subseteq E$$
$$V' = \{v \in V | (v, u) \in E' \text{for some } u\}$$

So we determine edge first, and then add nodes that the edge belong to.

7.1.3 Graph Isomorphism

$G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijection $f : V_1 \to V_2$ such that $(u, v) \in E_1$ iff $(f(u), f(v)) \in E_2$.

7.1.4 Subgraph Isomorphism

G_2 is subgraph-isomorphic to G_1 (G_1 is a subgraph of G_2) if some subgraph of G_2 is isomorphic to G_1

7.2 Network Motifs

7.2.1 Subgraph Frequency

7.2.1.1 Graph Level Frequency

Let G_Q be a small graph and G_T be a target graph. The graph-level subgraph frequency is defined as number of unique subsets of nodes V_T of G_T for which the subgraph of G_T induced by the nodes V_T is isomorphic to G_Q.

7.2.1.2 Node Level Frequency

Let G_Q be a small graph, v be a node in G_Q, and G_T be a target graph. The node-level subgraph frequency is defined as: The number of nodes u in G_T for which some subgraph of G_T is isomorphic to G_Q and the isomorphism maps to node u to v. (G_Q, v) is called a node-anchored subgraph.

7.2.2 Random Graph Generation

7.2.2.1 Erdos-Renyi(ER) random graphs

$G_{n,p}$ is an undirected graph on n nodes where each edge (u, v) appears iid with probability p

7.2.2.2 Configuration Model

The goal is to generate a random graph with a given degree sequence $k_1, ..., k_N$.

Method 1

- Generate nodes with spokes, and treat each node spoke as a mini-node
- Randomly pair up "mini" nodes. (Pick two from all the mini nodes that hasn't been paired, pair them)
- Connect node A to node B if there exists at least one mini-node connection.

The result will be a graph with approximately the same degree sequence as the given. Although in detail, we can have multiple mini-node connections between two nodes, or a inner-node mini-node connection, but the probability of those occurance gets really small for a large graph.

Method 2: Switching

- Start from a given graph G
- Repeat the switching step $Q \cdot |E|$ times. Each time, select a pair of edges $A \to B$ and $C \to D$ at random, and exchange the endpoints to give $A \to D$ and $C \to B$. We exchange edges only if no multiple edges of self-edges are generated.

7.2.3 Motif Significance

The overall process is

1. Count motifs in the given graph G_{real}
2. Generate a random graphs with similar statistics (number of nodes, edges, degree sequence and etc.), and count motifs in the random graphs
3. Use statistical measures to evaluate how significant is each motif (Z-score). Subgraph with high Z-score is the motif.

7.2.3.1 Z score and Network Significance Profile

The Z score for motif i is defined as

$$Z_i = \frac{N_i^{real} - \bar{N}_i^{rand}}{std(N_i^{rand}}$$

Network significance profile (SP) is defined as a vector of normalized z-scores:

$$SP_i = \frac{Z_i}{\sqrt{\sum_j Z_j^2}}$$

7.3 Neural Subgraph Representations and Counting

7.3.1 Setup

Given a large target graph (can be disconnected) and a query graph (connected), is query graph a subgraph in the target graph? We can use GNN to predict subgraph isomorphism by exploiting the geometric shape of embedding space to capture the properties of subgraph isomorphism.

Overall, we want to break the big graph into neighborhoods (containing node anchor), and compare the node-anchored embedding of the query graph and the neighborhoods.

7.3.2 Neighborhood Decomposition

- For each node in G_T:
  - Obtain a $k-hop$ neighborhood around the anchor. Can be performed using BFS, and k is a hyper-parameter
- For query G_Q, if the graph is large, do the same procedures. Otherwise, do not decompose
- Compute the embedding of the anchor nodes.

7.3.3 Order Embedding Space

We want to map graph A to a point z_A such that z_A is non-negative in all dimensions. Also, if B is a subgraph of A, then z_B has lower value for all coordinates compare to z_A.

7.3.4 Loss function to satisfy order embedding space

We specify the order constraint:

$$\forall_{i=1}^{D} \quad z_q[i] \leq z_u[i] \qquad iff \quad G_Q \subseteq G_T$$

GNN Embeddings are learned by minimizing a max-margin loss

$$E(G_q, G_t) = \sum_{i=1}^{D} (max(0, z_q[i] - z_t[i]))^2$$

7.3.5 Training Detail

Construct training examples (G_q, G_t) where half the time, G_q is a subgraph of G_t and the other half it is not. Train on those examples by minimizing the max-margin loss:

- For positive examples: Minimize $E(G_q, G_t)$
- For negative examples: Minimize $\max(0, \alpha - E(G_q, G_t))$

7.3.6 Training Example Construction

We can get G_T by choosing a random anchor v and taking all nodes in G within distance K from v to be in G_T

Positive examples: Sample induced subgraph G_Q of G_T with BFS sampling

- Initialize $S = \{v\}$, $V = \emptyset$
- Let $N(S)$ be all neighbors of nodes in S, at every step, sample 10% of the nodes in $N(S) - V$, put them in S. Put the remaining nodes of $N(S)$ IN v
- After K steps, take the sub-graph of G induced by S anchored at q

Negative examples: "Corrupt" G_Q by adding / removing nodes/edges so it is no longer a subgraph.

7.3.7 Prediction

Given query graph G_q anchored at node q and target graph G_t anchord at node t. Output whether the query is a node-anchored subgraph of the target by comparing $E(G_q, G_t)$. If $E(G_q, G_t) < \epsilon$, then predict true.

To check if G_Q is isomorphic to a subgraph of G_T, repeat the checking for all $q \in G_Q, t \in G_T$???

7.4 Mining Frequent Subgraphs

Goal: Find the frequency of the size k motifs.
SPminer: Estimate frequency of G_Q by counting the number of G_{N_i} such that their embeddings z_{N_i} staisfy $z_Q \leq z_{N_i}$

- Initial Step: Start by randomly picking a starting node u in the target graph G_T, set $S = \{u\}$
- Interatively: Grow a motif by iteratively choosing a neighbor in G_T of a node in S and add that node to S. We grow the motif S until it reaches the target size k

Chapter 8

Recommender Systems: Task and Evaluation

8.1 Top - K recommendation Setup

Recommender system can be modeled as a bipartie graph:

- Node: user U and item V
- Edge: user item interaction E

Challenge : Given the size of available user and item, even we have the true fuction f, we can not calculate it for every user - item pair. The solution is a two-stage process 1): Candidate generation and 2): Ranking.

Objective : Develop a real-valued scalar score $f(u, v)$ and the corresponding candidate generation and ranking process.

Metrics : Formally, we define our objective as the recall. For each user u, let P_u be a set of positive items that user will interact in the future. let R_u be a set of items recommended by the model, $|R_u| = K$. The metrics, recall, is defined as $(P_u \cap R_u)/P_u$. So what is the ratio between successfully identified product and the true set of positive items.

8.2 Embedding-Based Model Intuition

Create $D-$ dimensional embeddings for both user $u \in U$ and item $v \in V$. Let $f_\theta(\cdot, \cdot) : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be the parameterized scoring function.

8.3 Model Training Objective

Embedding-based models have three kinds of parameters:

- User embedding encoder
- Item embedding encoder
- Score function $f_\theta(\cdot, \cdot)$

8.4 Model Loss Function

Ideally we want to use the recall as the objective, but recall itself is not differentiable. So two surrogate loss functions are widely-used: Binary Loss and Bayesian Personalized Ranking (BPR) loss.

Binary Loss
Let positive edges to be the ones observed / training user-item interaction. Let the negative edges to be non existent edges (in reality it is tricky because non-existent edges doesn't mean it shouldn't exist. Some strategy exists to sample negative edges, such asProduct that user looked at but never bought). Let the sigmoid function be $\sigma(x) = \frac{1}{1+\exp(-x)}$. The binary loss is defined as

$$-\frac{1}{|E|}\sum_{(u,v)\in E}\log(\sigma(f_\theta(u,v))) - \frac{1}{|E_{neg}|}\sum_{(u,v)\in E_{neg}}\log(1-\sigma(f_\theta(u,v)))$$

The issue with binary loss is that the scores of ALL positive edges are pushed higher than those of ALL negative edges. This would penalize the model even if the model has perfect recall. Because at a user level, as long as that user's positive edge is higher than that user's negative edge, it is fine. It doesn't matter user A's positive edge has to have a higher score than user B's negative edge.

Bayesian Personalized Ranking (BPR) loss
For each user $u* \in U$, define the rooted positive/negative edges. The BPR loss for a specfic users is defined as

$$Loss(u*) = \frac{1}{|E|\,|E_{neg}|}\sum_{(u*,v_{pos})\in E(u*)}\sum_{(u*,v_{neg})\in E_{neg}(u*)} -\log(\sigma(f_\theta(u*,v_{pos}) - f_\theta(u*,v_{neg})))$$

The total loss is defined as the average BPR loss of all users. This way, we restrict the order of positive to negative edges on a user basis. During mini-batch training, for each user in the mini batch, we have one positive items, and a set of negative items. The negative items can be shared across users in mini batch.

8.5 Neural Graph Collaborative Filtering (NGCF)

8.5.1 Basic Framework

Given a bipartie graph, NGCF has two steps:

- For each user and item, prepare a shallow learnable embedding. For shallow embedding, we train it so that the score $f_\theta(u, v) \equiv z_u^T z_v$. Note that this score is maximized when there is an edge. So it only focuses on 1st order connections.
- Use multi-layer GNNs to propagate embedding along the bipartite graph. (One GNN to propagate user nodes, and one to propagate item nodes). The final score is the dot product of the scoring function.

8.6 Light GCN

Adjacency Matrix If we line up all the users, and then items, the adjacency matrix of a undirected bipartie graph can be viewed as

$$\begin{bmatrix} 0 & R \\ R^T & 0 \end{bmatrix}$$

Embedding Matrix We can have an embedding matrix E where the upper set of rows are user embedding and lower set of rows are item embedding.

GNN Aggregation Matrix Form Let D bt he degree matrix of A. The normalized adjacency matrix $\tilde{A} \equiv D^{-1/2}AD^{-1/2}$. Let $E^{(k)}$ be the embedding matrix at $k-th$ layer. Then each layer of GCN's aggregation can be written in a matrix form

$$E^{(k+1)} = ReLU(\tilde{A}E^{(k)}W^{(k)})$$

So if we remove the non-linearity, the final node embedding matrix is given as

$$E^{(K)} = \tilde{A}^K E W^{(0)} ... W^{(K-1)} = \tilde{A}^K E W$$

In this formulation, we can see that by applying $E = \tilde{A}E$ for K times, and then multiplying W, we can get the final embedding. This way we nver have to materlize $\tilde{A}^K$ which is a dense matrix.

Multi-scale Diffusion

$$\alpha_0 E^{(0)} + \alpha_1 E^{(1)} + ... + \alpha_K E^{(K)}$$

- $\alpha_k = \frac{1}{k+1}$

Light GCN Model Overview Given adjacency matrix A and initial learnable embedding matrix E, iteratively diffuse embedding E using $\tilde{A}$ for $k = 0, ..., K-1$. To get the final embedding, we take the average of embedding at every step. The scoring function is still the dot product. So the only learnable parameters are in the shallow embedding portion.

8.7 PinSAGE

Task: Recommend related pins to users.

8.7.1 PinSAGE Overview

PinSage is a GNN over the pins as well as the boards. So it takes into account not only the image attribute of the self pin, but also the image attribute of other pins that was shared in the same board, and or multiple hops away.

8.7.2 Curriculum Learning

At $n-$the epoch, we add $n-1$ hard negative items. For each user node, the hard negatives are item nodes that are close but not connected to the user node in the graph. It can be obtained as

- Compute personalized page rank for user u
- Sort items in the descending order of their PPR scores
- Randomly sample item nodes that are ranked in high but not too high, e.g.: 2000th - 5000th

This is specifically designed for Pinterest.

8.7.3 Negative Sampling

(q, p) positive pairs are given but various methods to sample negatives to form (p, q, n)

- Distance weighted sampling
- Hard negative mining
- Semi-hard negative mining
- Random

So PinSage uses a variety of negative sampling techniques. Also, given users in mini-batch, we want to use shared negative examples for all users to reduce data sampling.

Chapter 9

Community Detection

9.1 Premier

9.1.1 Structural Definition of Edge Overlap

$$O_{ij} = \frac{|(N(i) \cap N(j)) - \{i, j\}|}{|(N(i) \cup N(j)) - \{i, j\}|}$$

9.2 Louvain Algorithm

9.2.1 Null Model: Configuration Model

Given real G on n nodes and m edges, construct rewired network G' using the mini node method

- Same degree distribution but uniformaly random connections
- Consider G' as multigraph
- The expected number of edges between nodes i and j of degree k_i and k_j equals $\frac{k_i k_j}{2m}$ where m is the number of edges. ($2m$ number of mini nodes, k_i chance of connection, each connection there is $\frac{k_j}{2m}$ chance of connecting to j)

9.2.2 Modularity

Modularity is a measure of how well a network is partitioned into communities. Given a partitioning of the network into groups disjoint $s \in S$:

$$Q \propto \sum_{s \in S} [(\#\text{ edges within group s}) - (\text{expected }\#\text{ edges within group s})]$$

Note that this does not take into account edges between $s \in S$. The expected # of edges is determined by null model. The modularity of partitioning S of graph G can be simplified as

$$Q(G,S) = \frac{1}{2m} \sum_{s \in S} \sum_{i \in s} \sum_{j \in s} (A_{ij} - \frac{k_i k_j}{2m})$$

The range is $-1 \leq Q \leq 1$. This can also be extended to weighted edges. Generally, if modularity is greater than 0.3-0.7, it means significant community structure.

9.2.3 Louvain Algorithm Overview

Greedy algorithm for community detection with $O(n \log n)$ run time. It supports weighted graphs, and provides hierarchical communities. It initialize each node in a graph into a distinct community, then iterates two phases

- Phase 1: Modularity is optimized by allowing only local changes to node-communities memberships
- Phase 2: The identified communities are aggregated into super-nodes to build a new network. Edge weight between super-node is the weights sum of the edge weights in the original graph.

Phase 1 For each node i, the algorithm performs two calculations:

- Compute the modularity delta ∇Q when putting node i into the community of some neighbor j
- Move i to a community of node j that yields the largest gain in ∇Q

Repeat this process until no movement yields a gain.
∇Q if node i moves from community D to C can be decomposed into two terms: $\nabla Q(D \to i)$ and $\nabla Q(i \to C)$, i.e.: modularity change of removing i from D and

change of adding i to C.

$\nabla Q(i \to C)$

$$\sum_{in} \equiv \sum_{i,j \in C} A_{ij} \text{ Sum of link weights between nodes in C}$$

$$\sum_{tot} \equiv \sum_{i \in C} k_i \text{ Sum of all link weights of nodes in C}$$

$$Q(C)$$

$$= \frac{1}{2m} \sum_{i,j \in C} [A_{ij} - \frac{k_i k_j}{2m}]$$

$$= \frac{\sum_{i,j \in C} A_{ij}}{2m} - \frac{(\sum_{i \in C} k_i)(\sum_{j \in C} k_j)}{4m^2}$$

$$= \frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2$$

$$k_{i,in} \equiv \sum_{j \in C} A_{ij} + \sum_{j \in C} A_{ji} \text{ Sum of link weights connecting node } i \text{ and } C, \text{ double counted}$$

$$k_i \equiv \sum_{j} A_{ij} \text{ Sum of all link weights of node } i$$

$$Q_{before} = Q(C) + Q(\{i\}) = \frac{\sum_{in}}{2m} - (\frac{\sum_{tot}}{2m})^2 + [0 - (\frac{k_i}{2m})^2]$$

$$Q_{after} = Q(C + \{i\}) = \frac{\sum_{in} + k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_j}{2m}\right)^2$$

$$\nabla Q(i \to C) = Q_{after} - Q_{before}$$

$\nabla Q(D \to i)$ can be derived similarly.

9.3 Overlapping Community Detection: AGM

Two step:

- Define a generative model for graphs that is based on node community affiliations
- Given graph G, make the assumptionthat G was generated by AGM, and find the best AGM that could have generated G

9.3.1 AGM Detail

Model parameters: node V, Communities C, Membership M. For each community c has a single probability p_c to indicate the chance of edge within community members.

The communities C and nodes V can form a bipartite graph with membership M as edges. An edge exists indicates that node u belong to community c. Nodes in community c connect to each other by probability p_c. So $p(u,v) = 1 - \prod_{c \in M_u \cap M_v}(1-p_c)$

9.3.2 Fitting AGM via MLE

Given a graph, we need to calculate $p(G|F)$ where F is model parameters.

$$P(G|F) = \prod_{(u,v)\in G} p(u,v) \prod_{(u,v)\notin G} (1-p(u,v))$$

We want to to use $F = N \times C$ to represent membership association. F_{ij} is a non-negative score indicating how likely node i belong to community j. We can further model the edge probability as

$$P_c(u,v) = 1 - \exp(-F_{uC} \cdot F_{vC})$$

In the case of multiple communicty, we have

$$P(u,v) = 1 - \prod_{c\in\Gamma}(1-P_c(u,v)) = 1 - \exp(-F_u^T f_v)$$

So we want to maximize the log likelihood using the given $p(u,v)$ expression. We have

$$\sum_{(u,v)\in E} \log(1-\exp(-F_u^T F_v)) - \sum_{(u,v)\notin E} F_u^T F_v$$

9.4 Neural Overlapping Community Detection (NOCD)

Key idea: Generate F matrix using a GNN. e.g.:

$$F = GCN(A,X) = \sigma(\tilde{A}\sigma(\tilde{A}W_1)W_2)\tilde{A} = D^{-1}A$$

In order to deal with sparsity of real-world graphs,

$$l(F) = \frac{1}{|E|}\sum_{(u,v)\in E} \log(1-\exp(-F_u^T f_v)) - \frac{1}{n^2-|E|}\sum_{(u,v)\notin E} F_u^T F_v$$