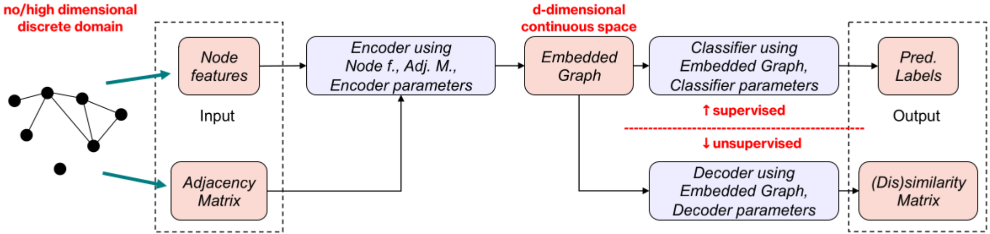
Skip-gram and Outer-product based graph shallow embedding methods

Yitao (Viola) Chen

Table of content

- Background
- Skip-gram methods
  - eg. DeepWalk, node2vec, WYS
- Matrix factorization methods
  - eg. Graph factorization (GF), GraRep, HOPE

Where we are in the full picture



What is Skip-gram

- Originally a method from natural language processing (NLP)
- Goal: learn word vector representations that are good at predicting the nearby word
- Example:

  The quick brown fox jumps over the lazy dog

  - lazy — dog, brown — fox
- car $\approx$ drive, car $\approx$ park, car $\not\approx$ boat (the likelihood of them appear in a close context is low, although they are "similar")

A few key things to note

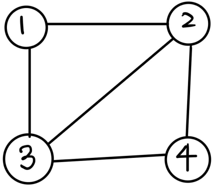
- how far the words are from each other doesn't matter (as long as it's in a specified window)
  - Example:

    The quick brown fox jumps over the lazy dog

  - suppose window size $= 5$, then ("jump" and "fox"), and ("jump" and "dog") are equally related
- this window is known as the context
- within context, word order doesn't matter

Skip-gram for graphs

- no "sentence" to begin with
- treat node as word
- generate sentences using random walk in the graph starting from each vertex



- e.g.

  random walks: [4, 2, 3], [4, 3, 2], [1, 2, 4]
- input: node feature $V \in \mathbb{R}^{|V| \times M}$ and adjacency matrix A
- output: learnt node embedding $\Phi \in \mathbb{R}^{|V| \times d}$

Why skip-gram

- Sometimes we care more about relational properties, ie. which nodes are connected and how connected are two nodes, without caring too much about their positional properties.
- random walk provides an overview of local connection around each vertex

DeepWalk

- set of truncated random walks of length $t \sim$ text corpus / sentences
- graph vertices $\sim$ vocabulary

* good to know vertex set V and frequency distribution of vertices in random walks ahead of training, but not necessary

Two components:

1. Random walk generator
2. update procedure through skip-gram

Algorithm overview

Algorithm 1 DeepWalk(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

```
1: Initialization: Sample Φ from U^{|V|×d}
2: Build a binary Tree T from V
3: for i = 0 to γ do
4:     O = Shuffle(V)
5:     for each v_i ∈ O do
6:         W_{v_i} = RandomWalk(G, v_i, t)
7:         SkipGram(Φ, W_{v_i}, w)
8:     end for
9: end for
```

Summary of algorithm

Initialize node embedding uniformly

for each round

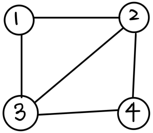
- permute nodes
- for each node in the permuted order
    - get a random walk
    - update embedding Φ

Return the learned embedding Φ

Random walk generator

Generate walk of length t starting from vertex v

- each walk samples uniformly from the neighbors of the last vertex visited until max length (t) is reached
- walks don't have to be of same length (could restart, but no obvious advantage in doing so)



e.g. $\mathcal{O} = \{4, 2, 1, 3\}$ and the first random walk from node 4 gives us $\mathcal{W}_{v_4} = [4, 2, 3]$ (walk length $t = 3$)

DeepWalk Skip-gram update procedure

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

for each $v_j \in \mathcal{W}_{v_i}$ **do**
    for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ **do**
        $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
        $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
    end for
end for

- given current representation of vertex $v_j, \Phi(v_j) \in \mathbb{R}^d$, want to maximize the probability of seeing its neighbors in the walk

Skip-gram loss function

- objective intuition: maximize the probability of seeing its close neighbors (those reachable within a number of steps) in the walk, given the node embedding $\Phi(v_i)$
- mathematically,

$$\max_{\Phi} \Pr(\{v_{i-w}, ...v_{i-1}, v_{i+1}, ..., v_{i+w}\} | \Phi(v_i))$$

  which is equivalent to maximizing sum of log probability (since order doesn't matter here)
- ie. $\max_{\Phi} \sum_{i-w \leq j \leq i+w, j \neq i} \log \Pr(v_j | \Phi(v_i))$
  which is not feasible to calculate since v_j is in the high dimensional feature space and $\Phi(v_i)$ is the low dimensional embedding
- solution: approximate $\Pr(v_j | \Phi(v_i))$

Computationally efficient ways to approximate softmax

- in NLP, softmax computation is impractical because the cost of computing $\Delta \log p(w_O|w_I)$ is proportional to $|W|$, the size in dictionary, which is often very large ($10^5 - 10^7$ terms in NLP problems)
- In graphs, calculating softmax is infeasible since we don't yet have ways to reason through the exact relationship between node feature vectors and their low-dimentional embeddings

There are ways to efficiently approximate the softmax:

- hierarchical softmax
- negative sampling / Noise contrastive estimation

References

Distributed Large-scale Natural Graph Factorization.
Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy.

GraRep: Learning Graph Representations with Global Structural information.
Shaosheng Cao, Wei Lu, Qiongkai Xu.

DeepWalk: Online Learning of Social Representations.
Bryan Perozzi, Rami Al-Rfou, Steven Skiena

Distributed Representations of Words and Phrases and their compositionality
Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean

Thank you for listening!

Yitao Chen

chen_yitao@gis.a-star.edu.sg