universidade de aveiro

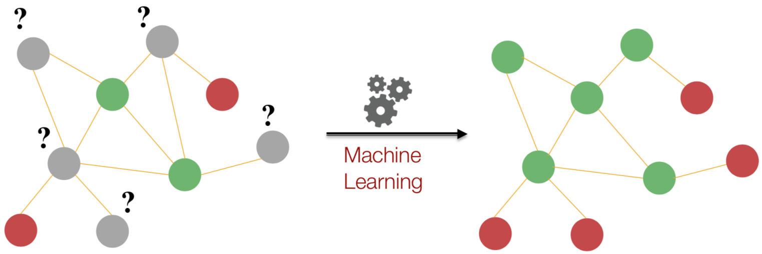
Mining Large Scale Datasets

Graph Representation Learning

(Adapted from CS246@Starford.edu; http://www.mmds.org)

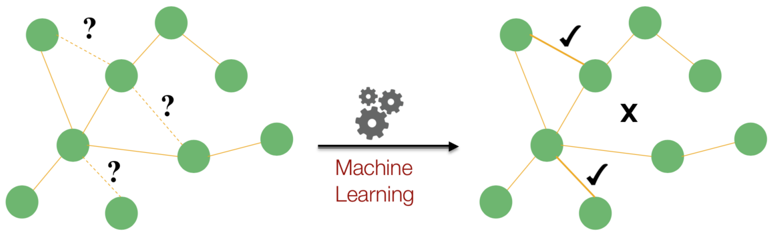
Sérgio Matos - aleixomatos@ua.pt

Machine Learning on Graphs



Node classification

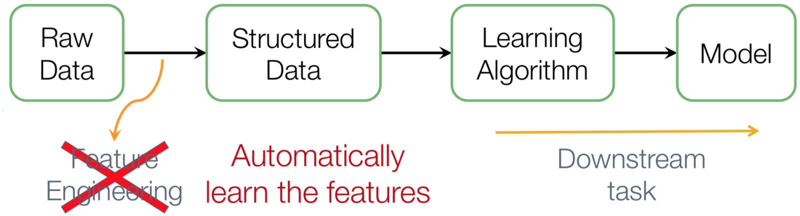
Machine Learning on Graphs



Link prediction

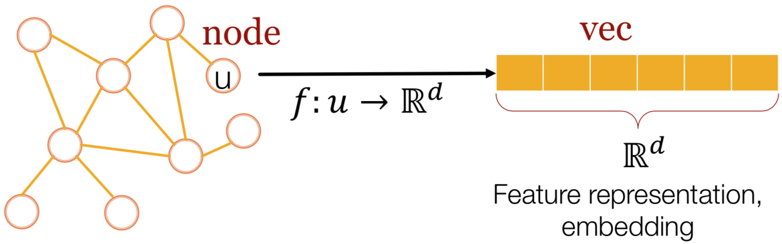
Machine Learning on Graphs

Machine Learning requires feature engineering!



Feature Learning on Graphs

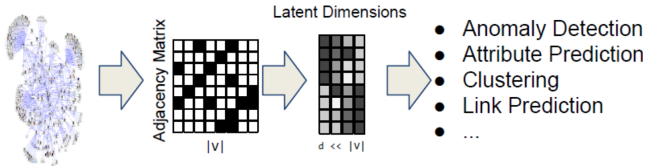
GOAL: Efficient task-independent feature learning for machine learning with graphs



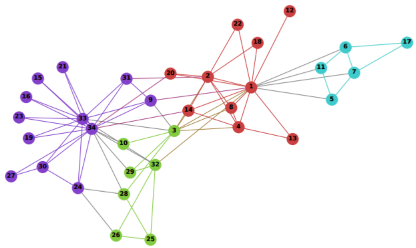
Feature Learning on Graphs

TASK: Map each node in a network into a low-dimensional space

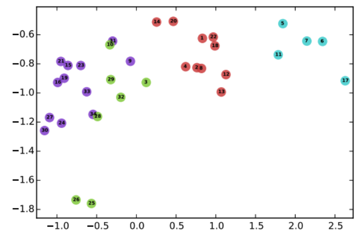
- Distributed representation of nodes
- Similarity of embeddings (node representations) between nodes indicates their network similarity
- Encode network information and generate node representation



Feature Learning on Graphs: Example



(a) Input: Karate Graph

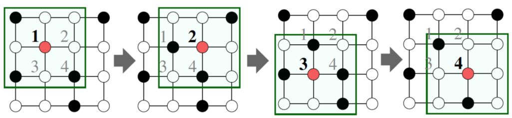


(b) Output: Representation

Image from: Perozzi et al. Deepwalk 2014

Feature Learning on Graphs: Differences

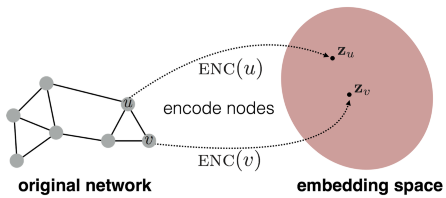
- Modern deep learning toolbox is designed for simple sequences or grids
  - CNNs for fixed-size images
  - RNNs or word2vec for text/sequences
- But networks are far more complex
  - Complex topographical structure: no spatial locality like grids
  - No fixed node ordering or reference point
  - Often dynamic and with multimodal features



Embedding graph nodes

- Assume a graph G
  - V is the vertex set
  - $\mathbf{A}$ is the (binary) adjacency matrix

- Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network

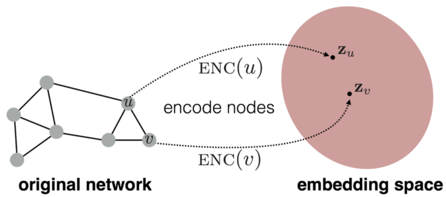


Embedding graph nodes

- Assume a graph G
  - V is the vertex set
  - $\mathbf{A}$ is the (binary) adjacency matrix

- Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the original network

$$\longrightarrow \boxed{\textit{similarity}(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u}$$



Embedding graph nodes: Steps

1. Define an encoder (i.e., a mapping from nodes to embeddings)
2. Define a node similarity function (i.e., a measure of similarity in the original network)
3. Optimize the parameters of the encoder so that

$$\mathit{similarity}(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u$$

Embedding graph nodes: Key components

- Encoder
  - Maps a node v to a low-dimensional vector $\mathbf{z}_v$

$$ENC(v) = \mathbf{z}_v$$

- Similarity function
  - Specifies how the relationships in vector space map to the relationships in the original network

$$similarity(u, v) \approx \mathbf{z}_v^\top \mathbf{z}_u$$

Shallow encoding

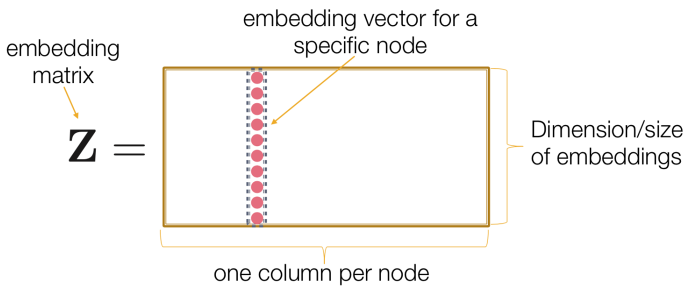
- Simplest encoding approach: encoder is just an embedding-lookup

$$ENC(v) = \mathbf{Z} \cdot \mathbf{v}$$

$\mathbf{Z} \in \mathbb{R}^{d \times \lvert\mathcal{V}\rvert}$	matrix, each column is a node embedding $\hookrightarrow$ <u>what we learn!</u>
$\mathbf{v} \in \mathbb{I}^{\lvert\mathcal{V}\rvert}$	indicator vector all 0s except a 1 in column indicating node v

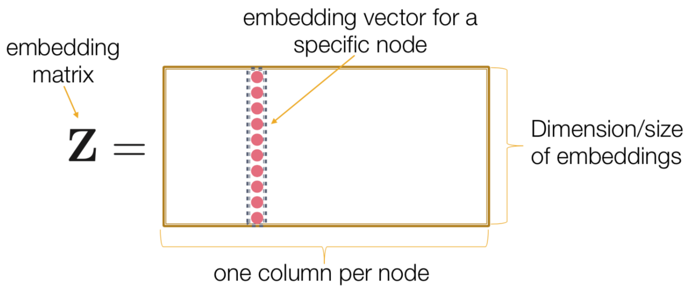
Shallow encoding

- Simplest encoding approach: encoder is just an embedding-lookup



Shallow encoding

- Simplest encoding approach: encoder is just an embedding-lookup



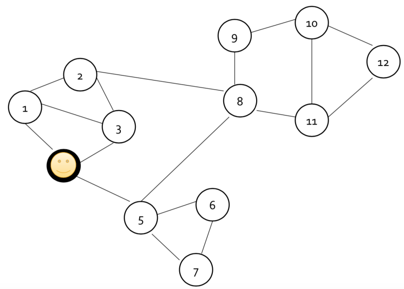
Methods: DeepWalk, node2vec, TransX, ...

Node similarity

- Key choice of methods is how they define node similarity

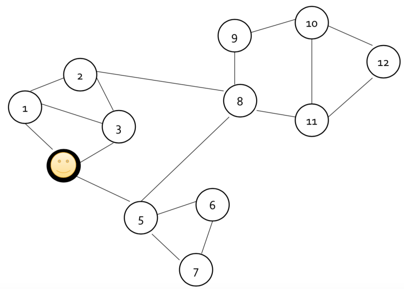
- E.g., should two nodes have similar embeddings if they ...
  - Are connected?
  - Share neighbours?
  - Have similar "structural roles"?
  - ...?

Random walk approaches



- Random walk on a graph
  - Given a starting point
  - Select a neighbor at random and move to that node
  - Repeat

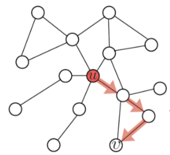
Random walk approaches



- Random walk on a graph
  - Given a starting point
  - Select a neighbor at random and move to that node
  - Repeat
- $\mathbf{z}_u^\top \mathbf{z}_v \approx$ probability that u and v co-occur on a random walk over the graph

Random walk embeddings

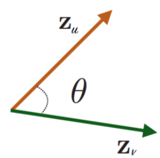
1. Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R



$P_R(v|u)$

2. Optimize embeddings to encode these random walk statistics

   Note: $\mathbf{z}_u^\top \mathbf{z}_v = cos(\theta)$ encodes the random walk 'similarity'



$\theta \propto P_R(v|u)$

Random walks

- **Expressivity**

  Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information

- **Efficiency**

  No need to consider all node pairs when training

  Only need to consider pairs that co-occur on random walks

Unsupervised feature learning

Intuition: Find d-dimensional embedding of nodes that preserves similarity

Idea: Learn node embedding such that nearby nodes are close together in encoding space

Given a node u how do we define nearby nodes?

$N_R(u)$: neighbourhood of u obtained by some strategy R

Feature learning as Optimization

Given $G = (V, E)$

Our goal is to learn a mapping $z : u \longrightarrow R^d$

Log-likelihood objective:

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

where $N_R(u)$ is the neighborhood of node u by strategy R

Given node u we want to learn feature representations that are predictive of the nodes in its neighborhood $N_R(u)$

Random Walk Optimization

1. Run short fixed-length random walks starting from each node on the graph using some strategy R
2. For each node u collect $N_R(u)$, the multiset of nodes visited on random walks starting from u

   Note: $N_R(u)$ can have repeated elements since nodes can be visited multiple times on random walks
3. Optimize embeddings according to

   (Given node u, predict its neighbors $N_R(u)$)

$$\max_{z} \sum_{u \in V} log P(N_R(u)|z_u)$$

Random Walk Optimization

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -log(P(v|\mathbf{z}_u))$$

- **Intuition**

  Optimize embeddings to maximize likelihood of random walk occurrences

- Parameterize $P(v|\mathbf{z}_u)$ using softmax

$$P(v|\mathbf{z}_u) = \frac{exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \mathbf{z}_u^\top \mathbf{z}_n}$$

Random Walk Optimization

$$\mathcal{L} = \underbrace{\sum_{u \in V}}_{\text{sum over all nodes } u} \underbrace{\sum_{v \in N_R(u)}}_{\text{sum over nodes } v \text{ seen on random walks starting from } u} - \log \left(\underbrace{\frac{\exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^\top \mathbf{z}_n)}}_{\text{predicted probability of } u \text{ and } v \text{ co-occuring on random walk}} \right)$$

Optimizing random walk embeddings = finding embeddings $\mathbf{z}_u$ that minimize $\mathcal{L}$

Random Walk Optimization

But doing this naively is too expensive!!

$$\mathcal{L} = \sum_{\boxed{u \in V}} \sum_{v \in N_R(u)} -\log\left(\frac{\exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{\boxed{n \in V}} \exp(\mathbf{z}_u^\top \mathbf{z}_n)}\right)$$

Nested sum over nodes means $O(|V|^2)$ complexity!

Negative Sampling

- **Solution**: Negative sampling

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

$$log\left(\frac{exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \mathbf{z}_u^\top \mathbf{z}_n}\right) \approx log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - \sum_{i=1}^{k} log(\sigma(\mathbf{z}_u^\top \mathbf{z}_{n_i})), n_i \sim P_V$$

σ: sigmoid function, makes each term a "probability" between 0 and 1

P_V: random distribution over all nodes

Negative Sampling

$$log\left(\frac{exp(\mathbf{z}_u^\top \mathbf{z}_v)}{\sum_{n \in V} \mathbf{z}_u^\top \mathbf{z}_n}\right) \approx log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - \sum_{i=1}^{k} log(\sigma(\mathbf{z}_u^\top \mathbf{z}_{n_i})), n_i \sim P_V$$

- Sample k negative nodes proportionally to degree
- Two considerations for k (number of negative samples):
  1. Higher k gives more robust estimates
  2. Higher k corresponds to higher prior on negative events
     In practice $k = 5..20$

Random Walks: overview

1. Run short fixed-length random walks starting from each node on the graph using some 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 using Stochastic Gradient Descent (approximating through negative sampling)

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -log(P(v|\mathbf{z}_u))$$

Walk this (or that) way...

- We have described how to optimize embeddings given random walk statistics
- What strategies should we use to run these random walks?
  - DeepWalk (Perozzi et al., 2013): Fixed-length, unbiased random walks starting from each node
  - This notion of similarity is too constrained
- How can we generalize this?

node2vec: Overview

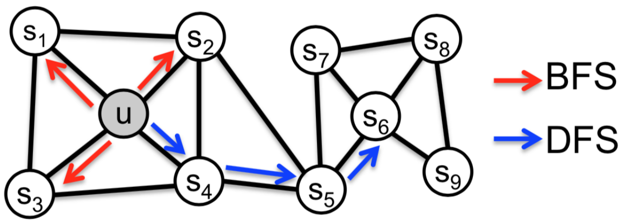
- Goal: Embed nodes with similar network neighborhoods close in the feature space
  - Framed as a maximum likelihood optimization problem, independent of the downstream prediction task

- Key observation: Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings

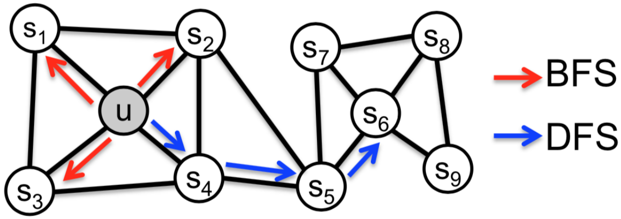
- node2vec: create biased second order random walk R to generate network neighborhood $N_R(u)$ of node u

node2vec: biased walks

- **Idea**: use flexible, biased random walks that can trade off between local and global views of the network



node2vec: biased walks



- Walk of length 3 (or neighborhood $N_R(u)$ 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

node2vec: interpolating BFS and DFS

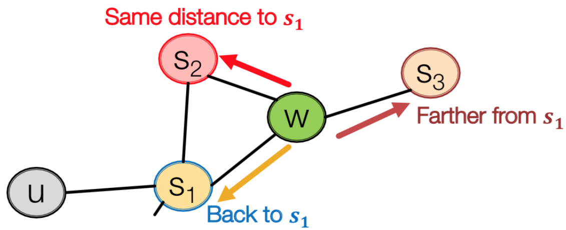
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- 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

node2vec: biased random walks

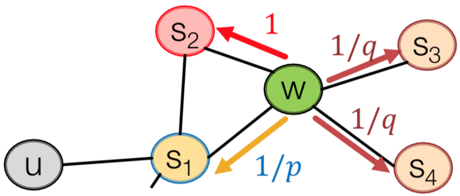
Biased 2^{nd}-order random walks explore network neighborhoods

Random walk just traversed edge (s_1, w) and is now at w
Neighbors of w can only be



node2vec: biased random walks

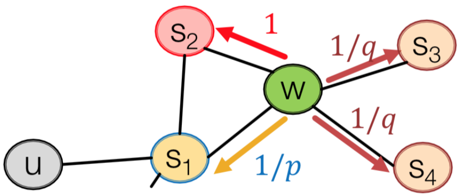
Random walk just traversed edge (s_1, w) and is now at w
Where to go next?



- p, q: model transition probabilities
  - p return parameter
  - q "walk away" parameter
- Note: $1/p$, $1/q$, 1, are unnormalized probabilities

node2vec: biased random walks

Random walk just traversed edge (s_1, w) and is now at w
Where to go next?



$w \rightarrow$

Target t	Prob.	Dist. (s_1, t)
s_1	$1/p$	0
s_2	1	1
s_3	$1/q$	2
s_4	$1/q$	2

- **BFS-like** walk: low value of p
- **DFS-like** walk: low value of q

node2vec algorithm

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

Linear-time complexity
All 3 steps are individually parallelizable

How to use the node embeddings

- **Clustering/community detection**: Cluster points z_i
- **Node classification**: Predict label $f(z_i)$ of node based on z_i
- **Link prediction**: Predict edge (i, j) based on $f(z_i, z_j)$
  - Where we can apply:

    Concatenation: $f(z_i, z_j) = g([z_i, z_j])$

    Hadamard product: $f(z_i, z_j) = g(z_i \odot z_j)$

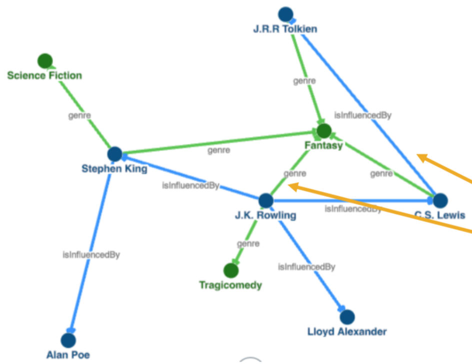
    Sum/Average: $f(z_i, z_j) = g([z_i, z_j])$

    Distance: $f(z_i, z_j) = g(||z_i - z_j||_2)$

Summary

- **Basic idea**: Embed nodes so that distances in embedding space reflect node similarities in the original network
- Different notions of node similarity
  - Adjacency-based (i.e., similar if connected)
  - Multi-hop similarity definitions
  - Random walk approaches (covered today)
- No single method wins in all cases...
  - e.g. node2vec performs better on node classification while multi-hop methods performs better on link prediction
- Random walk approaches are generally more efficient
- **In general**: Must choose definition of node similarity that matches your application!

Knowledge Graph Embeddings

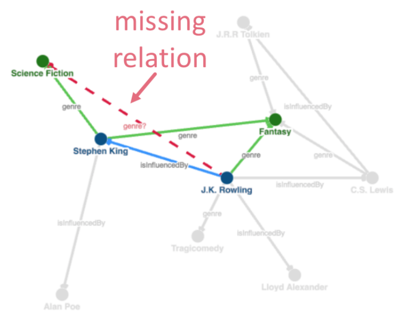


A **knowledge graph** is composed of facts/statements about inter-related entities

In KGs, edges can be of many types!

Nodes are referred to as **entities**, edges as **relations**

Knowledge Graph completion



KG **incompleteness** can substantially affect the efficiency of systems relying on it!

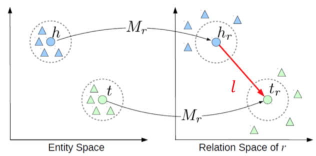
Create a link prediction model that learns from local and global connectivity patterns in the KG, taking into account <u>entities and relationships of different types</u> at the same time

TransE

- In TransE, relationships between entities are represented as triplets
  (head entity), (relation), (tail entity) : (h, l, t)

- Entities are first embedded in an entity space R^k
  - similarly to the previous methods

- Relations are represented as **translations**
  - $h + l \approx t$ if the fact is true
  - else, $h + l \neq t$



TransE algorithm

Algorithm 1 Learning TransE

input Training set $S = \{(h, \ell, t)\}$, entities and rel. sets E and L, margin γ, embeddings dim. k.

initialize $\boldsymbol{\ell} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each $\ell \in L$
$\boldsymbol{\ell} \leftarrow \boldsymbol{\ell} / \|\boldsymbol{\ell}\|$ for each $\ell \in L$
$\mathbf{e} \leftarrow \text{uniform}(-\frac{6}{\sqrt{k}}, \frac{6}{\sqrt{k}})$ for each entity $e \in E$

Entities and relations are initialized uniformly, and normalized

loop
  $\mathbf{e} \leftarrow \mathbf{e} / \|\mathbf{e}\|$ for each entity $e \in E$
  $S_{batch} \leftarrow$sample(S, b) // sample a minibatch of size b
  $T_{batch} \leftarrow \emptyset$ // initialize the set of pairs of triplets
  for $(h, \ell, t) \in S_{batch}$ **do**
    $(h', \ell, t') \leftarrow$sample$(S'_{(h,\ell,t)})$ // sample a corrupted triplet

Negative sampling with triplet that does not appear in the KG

    $T_{batch} \leftarrow T_{batch} \cup \{((h, \ell, t), (h', \ell, t'))\}$
  end for
  Update embeddings w.r.t. $\displaystyle\sum_{((h,\ell,t),(h',\ell,t')) \in T_{batch}} \nabla\big[\gamma + \underbrace{d(\boldsymbol{h} + \boldsymbol{\ell}, \boldsymbol{t})}_{\text{positive sample}} - \underbrace{d(\boldsymbol{h'} + \boldsymbol{\ell}, \boldsymbol{t'})}_{\text{negative sample}}\big]_+$
end loop

Comparative loss: favors lower distance values for valid triplets, high distance values for corrupted ones