Deep learning techniques for Graph Embeddings

Yannis Kotidis

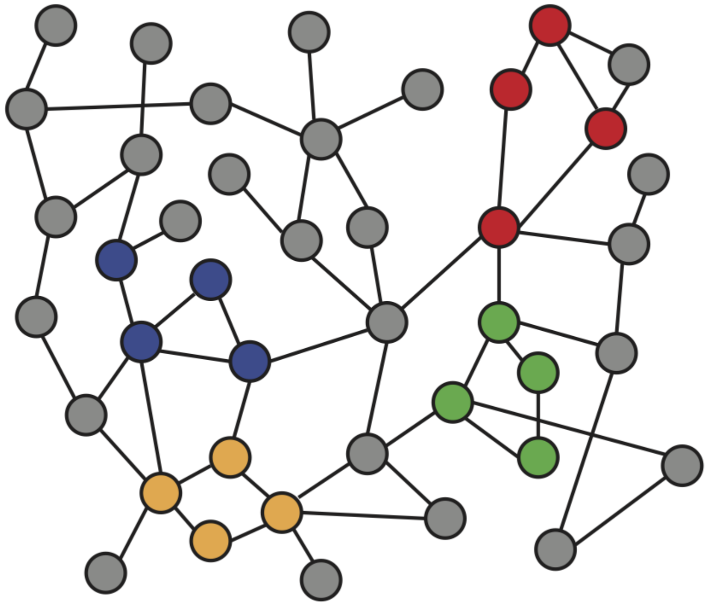
Acknowledgements

- Some of the presented material adapted from the following sources:
  - ISMB 2018 Tutorial on Deep Learning for Network Biology (http://snap.stanford.edu/deepnetbio-ismb/)
  - DeepWalk: Online Learning of Social Representations, Bryan Perozzi, Rami Al-Rfou, Steven Skiena, Stony Brook University KDD 2014
  - https://towardsdatascience.com/overview-of-deep-learning-on-graph-embeddings-4305c10ad4a4
  - http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

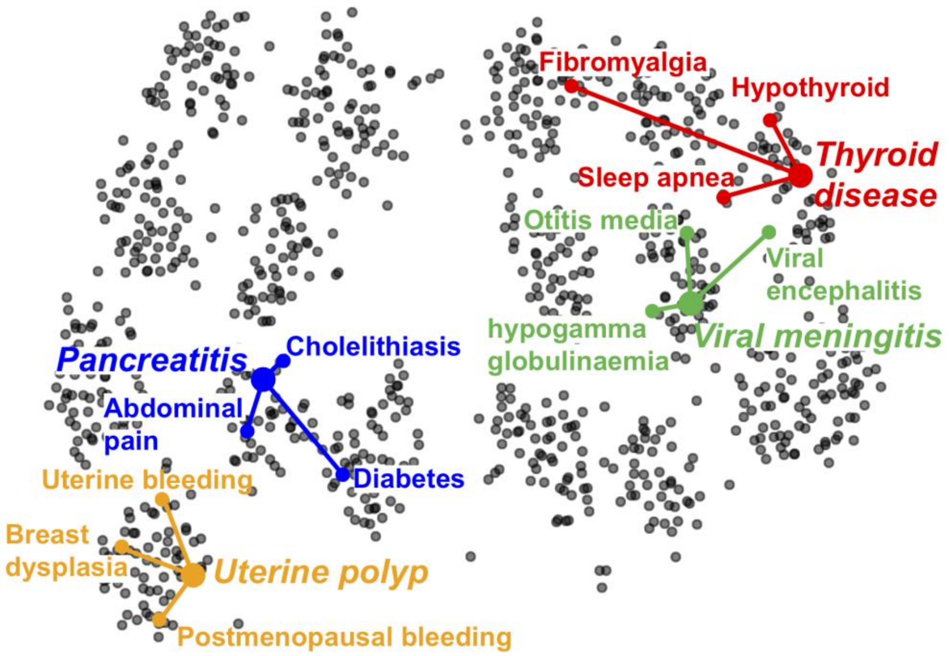
Motivation

- Pre-processing step in order to turn a graph into a computationally digestible format.
- Many Data Mining and Machine learning algorithms are tuned for continuous data mapped in a d-dimensional space.
- Visualization, outlier detection, etc.

Node embeddings: intuition



Input



Output

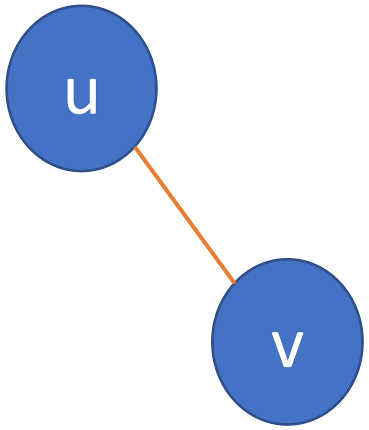
Map nodes to d-dimensional space such that similar nodes in the graph are embedded close together

Embedding methods

- Several existing methods:
  - node2vec, DeepWalk, LINE, struc2vec
- These techniques extract topological features in the form of common neighbors, paths, random walks, rooted trees, etc in order capture different notions of node similarity
- They utilize these features in order to embed graph nodes in a d-dimensional space

Simple Idea: two nodes are similar if they are connected

- Let **A** be the adjacency matrix for the graph
  - Then $A_{u,v}=1$ iff there is an edge between nodes u,v

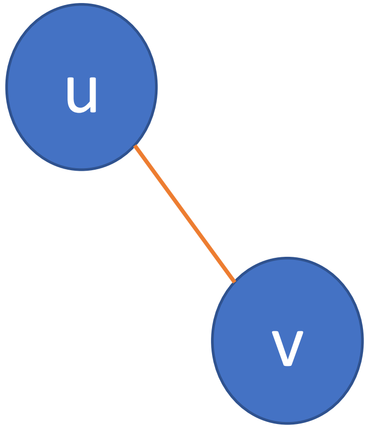


- Let z_u, z_v be the n-dim vector representations of these nodes, respectively
  - Let $z_u^T z_v$ denote their similarity (inner product)
  - We seek representations such that:

$$z_u^{T} \cdot z_v \approx \mathbf{A}_{\mathbf{u,v}}$$

Simple Idea: two nodes are similar if they are connected

- Let **A** be the adjacency matrix for the graph
  - Then $A_{u,v}=1$ iff there is an edge between nodes u,v



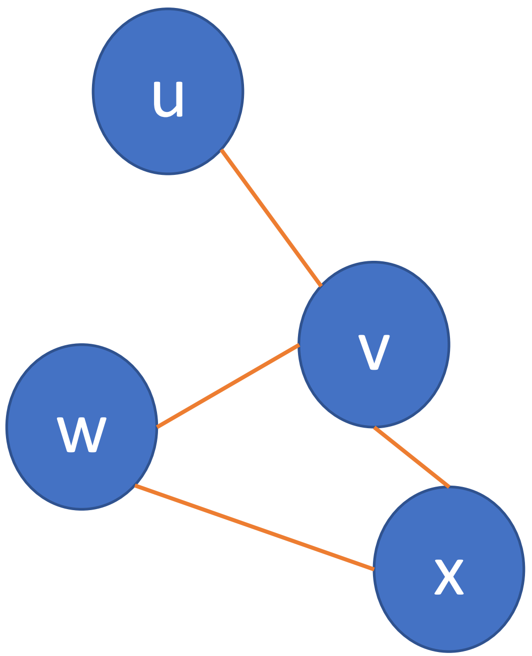
- Let z_u, z_v be the n-dim vector representations of these nodes, respectively
  - Let $z_u^T z_v$ denote their similarity (inner product)
  - We seek representations such that:

$$z_u^{T} \cdot z_v \approx \mathbf{A_{u,v}}$$

Trivial for two nodes:
Zu=(0,1,0)
Zv=(0,1,0)

Simple Idea: two nodes are similar if they are connected

- Let **A** be the adjacency matrix for the graph
  - Then $A_{u,v}$=1 iff there is an edge between nodes u,v
  - May use edge-weight for weighted graphs
- Let z_u, z_v be the n-dim vector representations of these nodes, respectively
  - Let $z_u^T z_v$ denote their similarity (inner product)
  - We seek representations such that:



$$z_u^{T} \cdot z_v \approx \mathbf{A_{u,v}}$$

Zu=(0,1,0)
Zv=(0,1,0)

Now what?

Adjacency-based Similarity

- **Similarity function** is the edge weight between *u* and *v* in the network
- **Intuition:** Dot products between node $z_u^{T} \cdot z_v$ embeddings approximate edge existence

Can be solved using Stochastic gradient descent (SGD)

$$\mathcal{L} = \sum_{(u,v) \in V \times V} \| \mathbf{z}_u^\top \mathbf{z}_v - \mathbf{A}_{u,v} \|^2$$

loss (what we want to minimize)

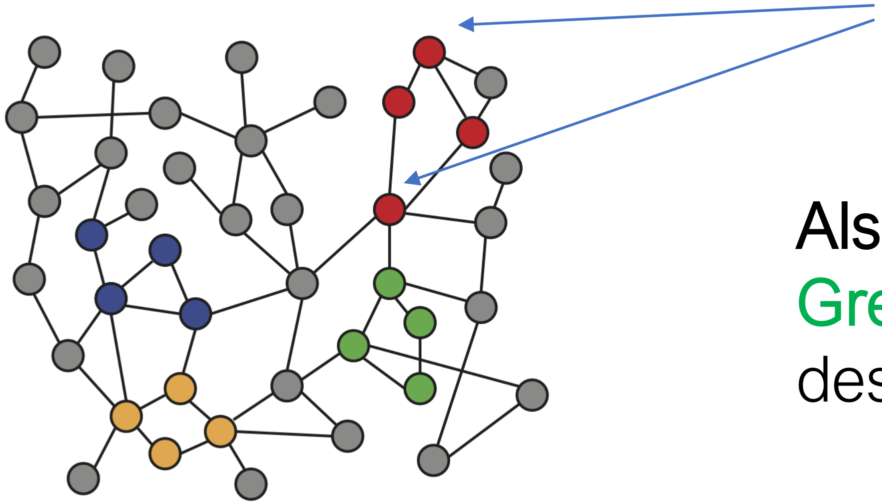
sum over all node pairs

embedding similarity

(weighted) adjacency matrix for the graph

Adjacency-based Similarity Shortcomings

- Complexity: must consider all node pairs →$O(|V|^2)$ runtime
- Only considers direct connections (example bellow)

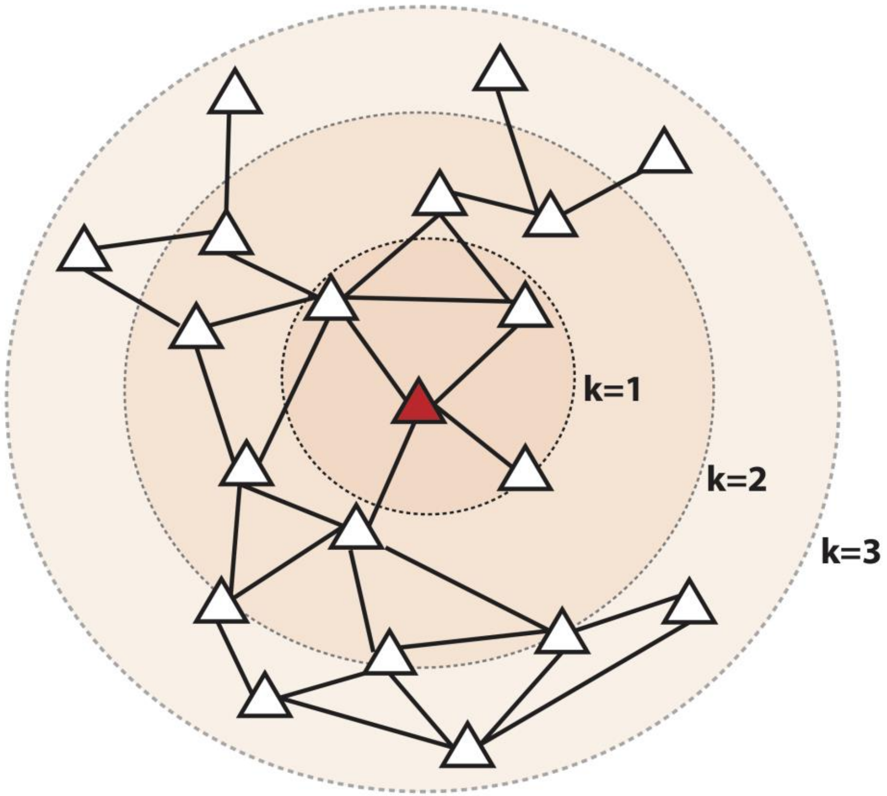


These two nodes are dissimilar per our definition of similarity

Also, we expect red nodes be more similar to Green nodes compared to Orange nodes, despite none being directly connected

Multi-Hop Similarity

Idea: Define node similarity function based on higher-order neighborhoods



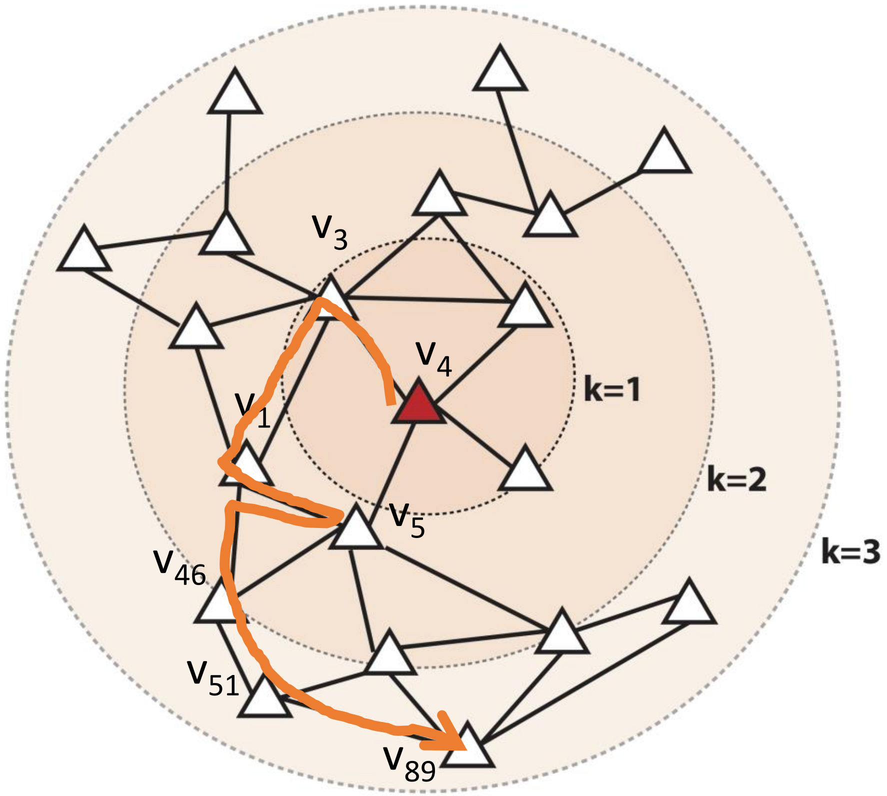
- Red: Target node
- k=1: 1-hop neighbors
  - i.e., adjacency matrix $\mathbf{A}$
- k=2: 2-hop neighbors
- k=3: 3-hop neighbors

How to stochastically define these higher-order neighborhoods?

→ Random Walks

Random Walk Example

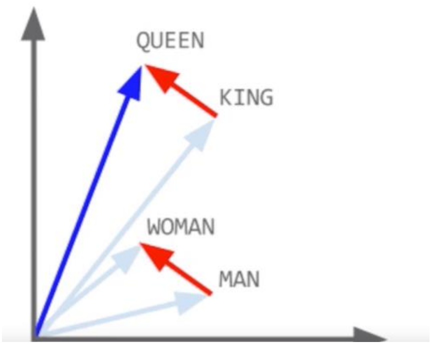
- Start from source node v_4 and walk for a while following graph edges
- Collected path: $v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_1 \rightarrow v_{46} \rightarrow v_{51} \rightarrow v_{89}$



- Intuition: nodes are similar if they are "close" in the network topology
- Such nodes frequently co-occur in a random walk
  - We will learn embeddings that will boost the similarity between such nodes

Word2vec

- Popular technique for creating vector representations of words
  - Intuition: Two words are similar if the frequently appear in the same context
    - Same context ≈ *within small distance in the same sentence*
  - *Is believed to capture both syntactic and semantic* relationships between words:
    - *Let Φ(x) be the learnt representation (vector) of word x*
    - *Φ("King") - Φ("Man") + Φ("Woman") ≈ Φ(“Queen“)*

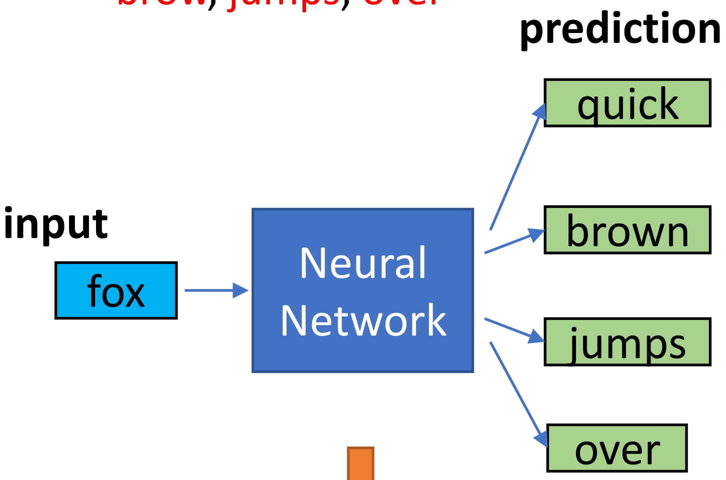


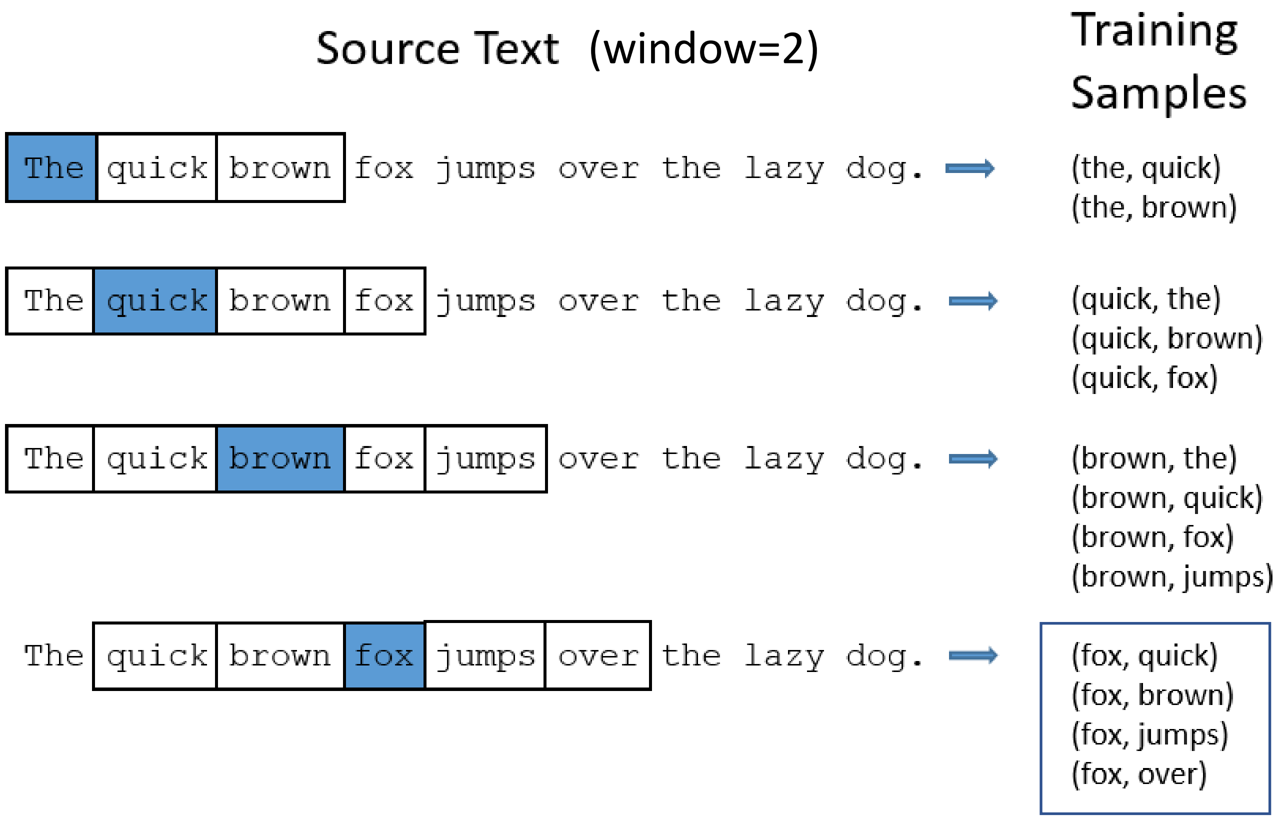
More examples (from product descriptions in online catalogs):
(https://medium.com/arvind-internet/applying-word2vec-on-our-catalog-data-2d74dfee419d)

- **shirt — buttons ≈ sweater**
- **suit — shirt — bow — waistcoat ≈ jeans**
- **party + weekend + clothing ≈ holiday**

The Skip-gram model

- Given an input word try to **predict** the **previous** w and **following** w words (w = window size)
- In the last training sample for input = fox try to predict quick, brow, jumps, over





Learn a representation Φ("fox") that maximizes these probabilities:

P["quick" |Φ("fox")]
P["brown" |Φ("fox")]
P["jumps" |Φ("fox")]
P["over" |Φ("fox")]

for this training input

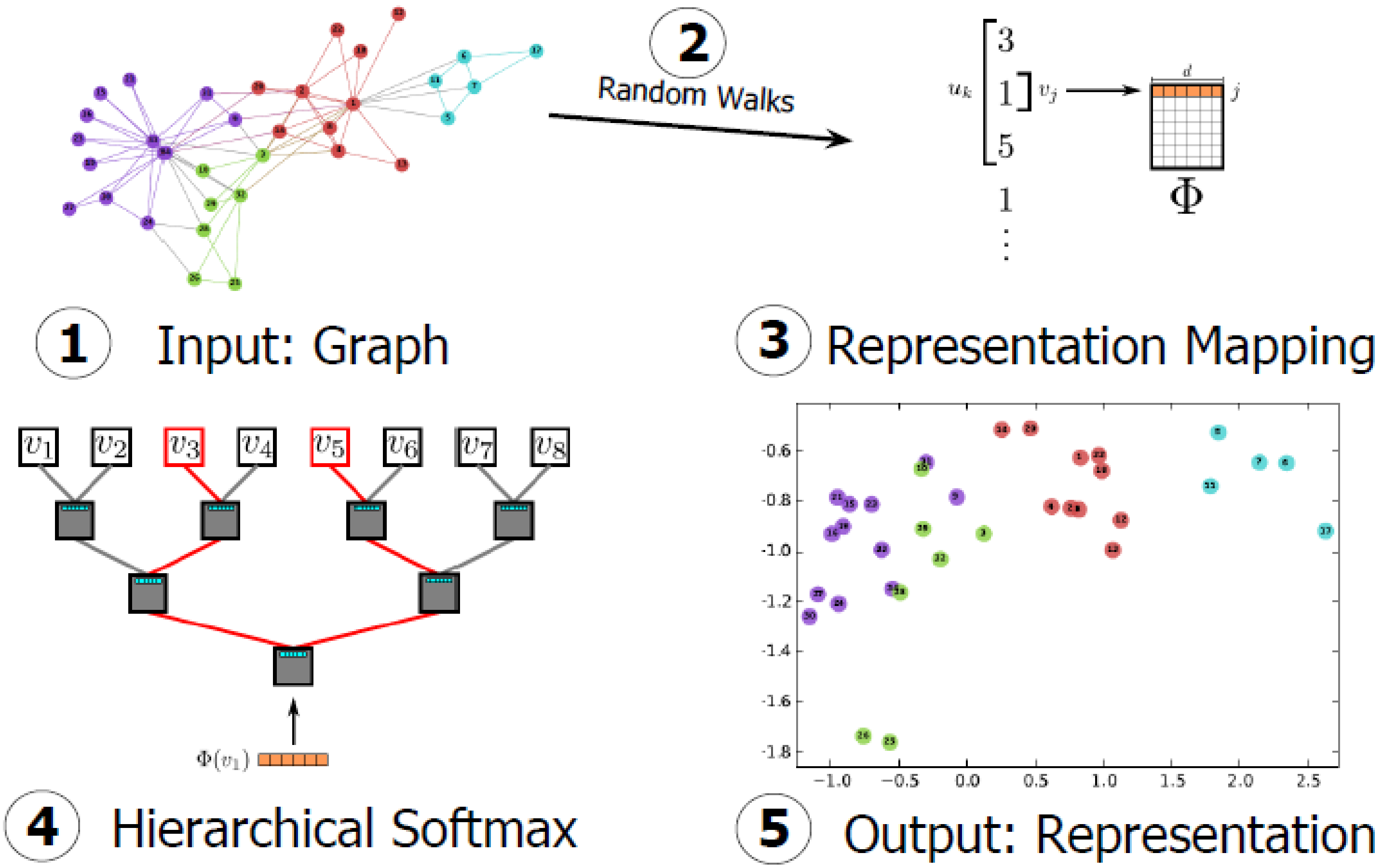
Neat Idea (Deep Walk)

- In the previous discussion replace
  - Words with graph nodes
  - Sentences with node sequences from short random words
- Observation
  - Words frequency in a natural language corpus follows a power law
  - Vertex frequency in random walks on scale free graphs also follows a power law
- Advantages
  - Flexibility: captures local and higher-order neighborhoods
  - Efficiency: Do not need to consider all node pairs when training
    - Consider only node pairs that co-occur in random walks

Deep Walk Framework

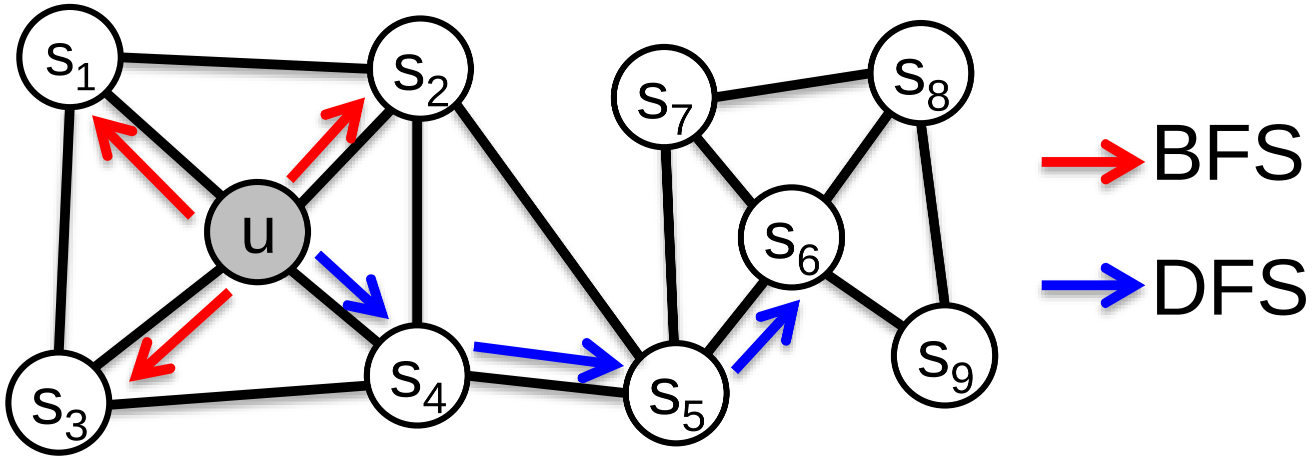
window =1

$v_4 \to v_3 \to v_1 \to v_5 \to v_1 \to v_{46} \to v_{51} \to v_{89}$



node2vec: Biased Walks

Two classic strategies to define a neighborhood $N_R(u)$ of a given node u:



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

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

Interpolate BFS and DFS

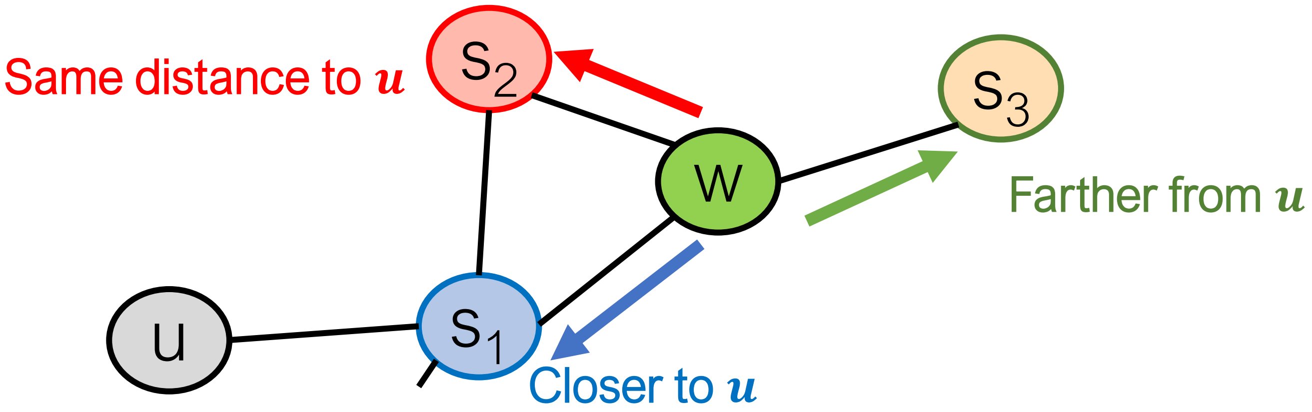
Biased random walk R that given a node u generates neighborhood $N_R(u)$

- Two parameters:
  - Return parameter p:
    - Return back to the previous node
  - In-out parameter q:
    - Moving outwards (DFS) vs. inwards (BFS)

Biased Random Walks

Biased 2nd-order random walks explore network neighborhoods:

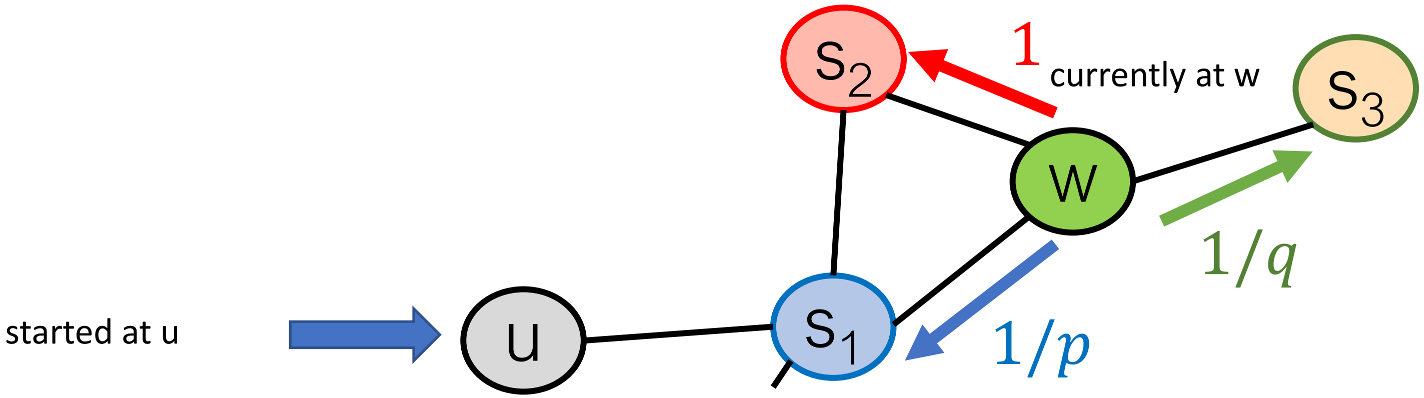
- Rnd. walk started at u and is now at w
- **Insight:** Neighbors of w can only be:



Idea: Remember where that walk came from

Biased Random Walks

- Walker is at **W**. Where to go next?

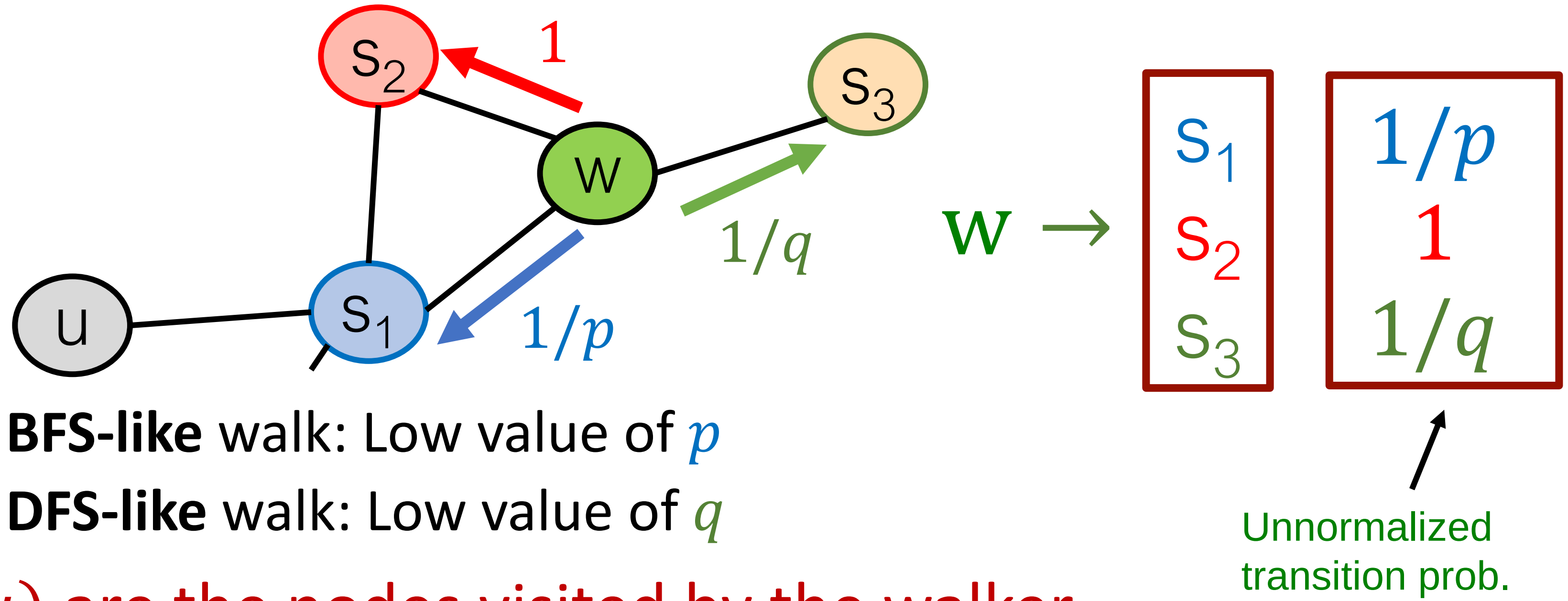


$1/p, 1/q, 1$ are unnormalized probabilities

- p, q model transition probabilities
  - p … “return” parameter (lower values are preferable)
  - q … ”walk away” parameter (lower values are preferable)

Biased Random Walks

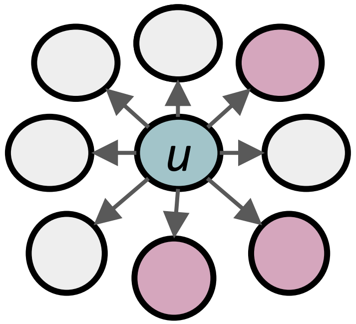
- Walker is at **W**. Where to go next?



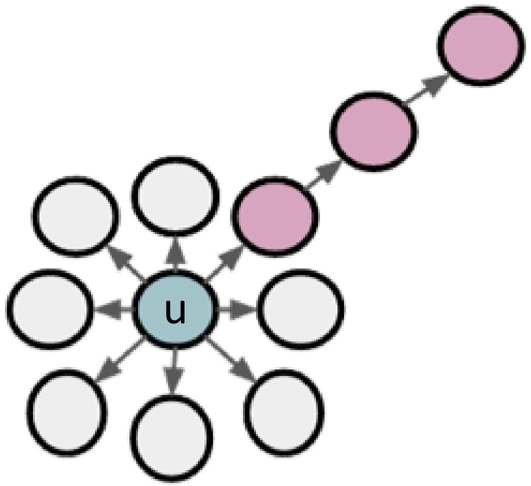
- **BFS-like** walk: Low value of p
- **DFS-like** walk: Low value of q

$N_S(u)$ are the nodes visited by the walker

BFS vs. DFS



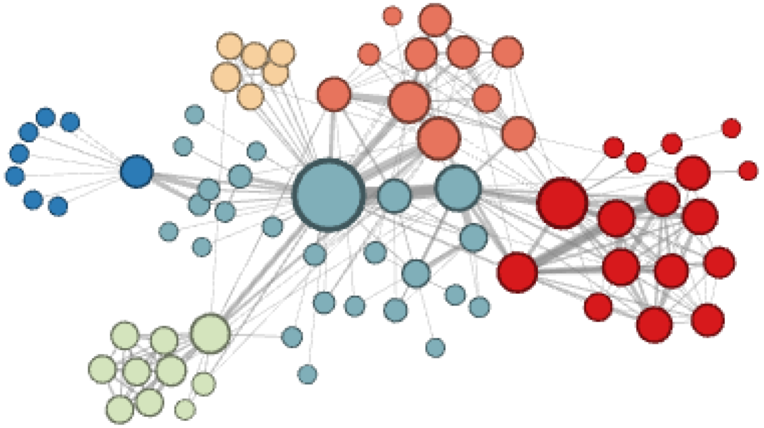
BFS:
Micro-view of
neighbourhood



DFS:
Macro-view of
neighbourhood

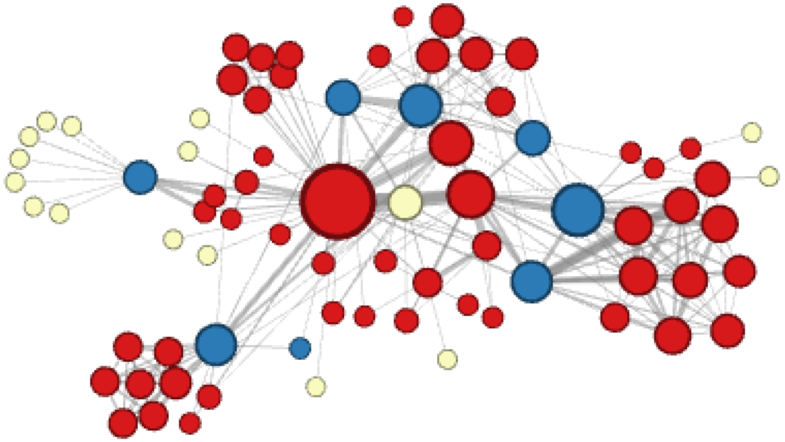
Experiment: Micro vs. Macro

Interactions of characters in a novel:



p=1, q=2

Microscopic view of the network neighbourhood

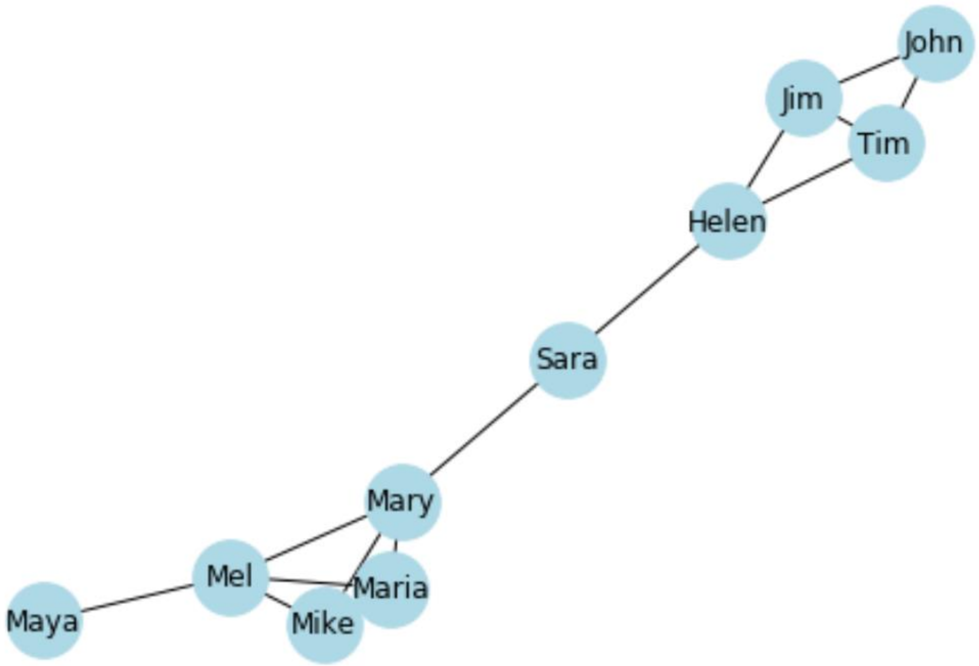


p=1, q=0.5

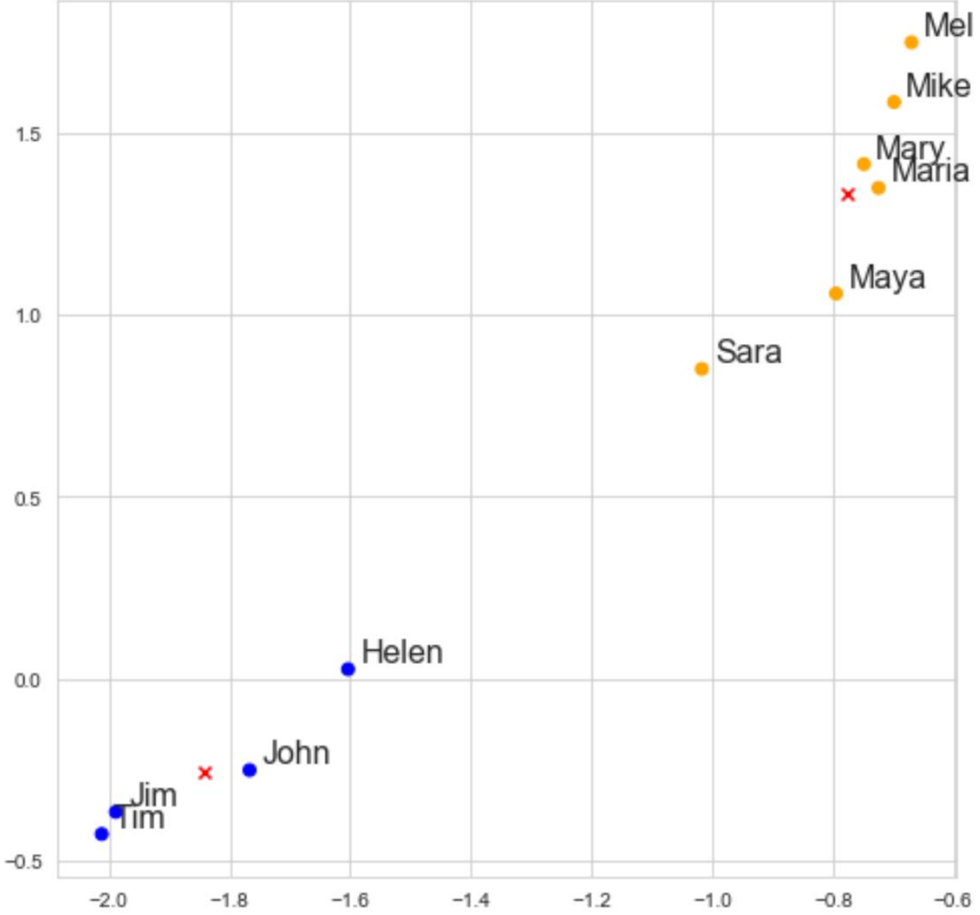
Macroscopic view of the network neighbourhood

Node2vec example

Input network



Clustering of resulting 2-dim vectors (p=1,q=2,w=3) with k-Means (k=2)



Graph Convolutional Networks

GCNs application

- **Semi-supervised learning:** Given a single network with partial nodes being labelled and others remaining unlabelled, GCN's model can identify the class labels for the unlabelled nodes
- **Graph node embedding:** We can use GCNs to represent each node as an aggregate of its neighbourhood and derive node embeddings

- For more details see https://arxiv.org/pdf/1609.02907.pdf

What is Convolution (image processing)

- Try to learn from the provided image by computing weighted averages of pixel values of the red pixel along with its neighbours
- Pass the computed result to an activation function that propagates the result to the next layer of the CNN.

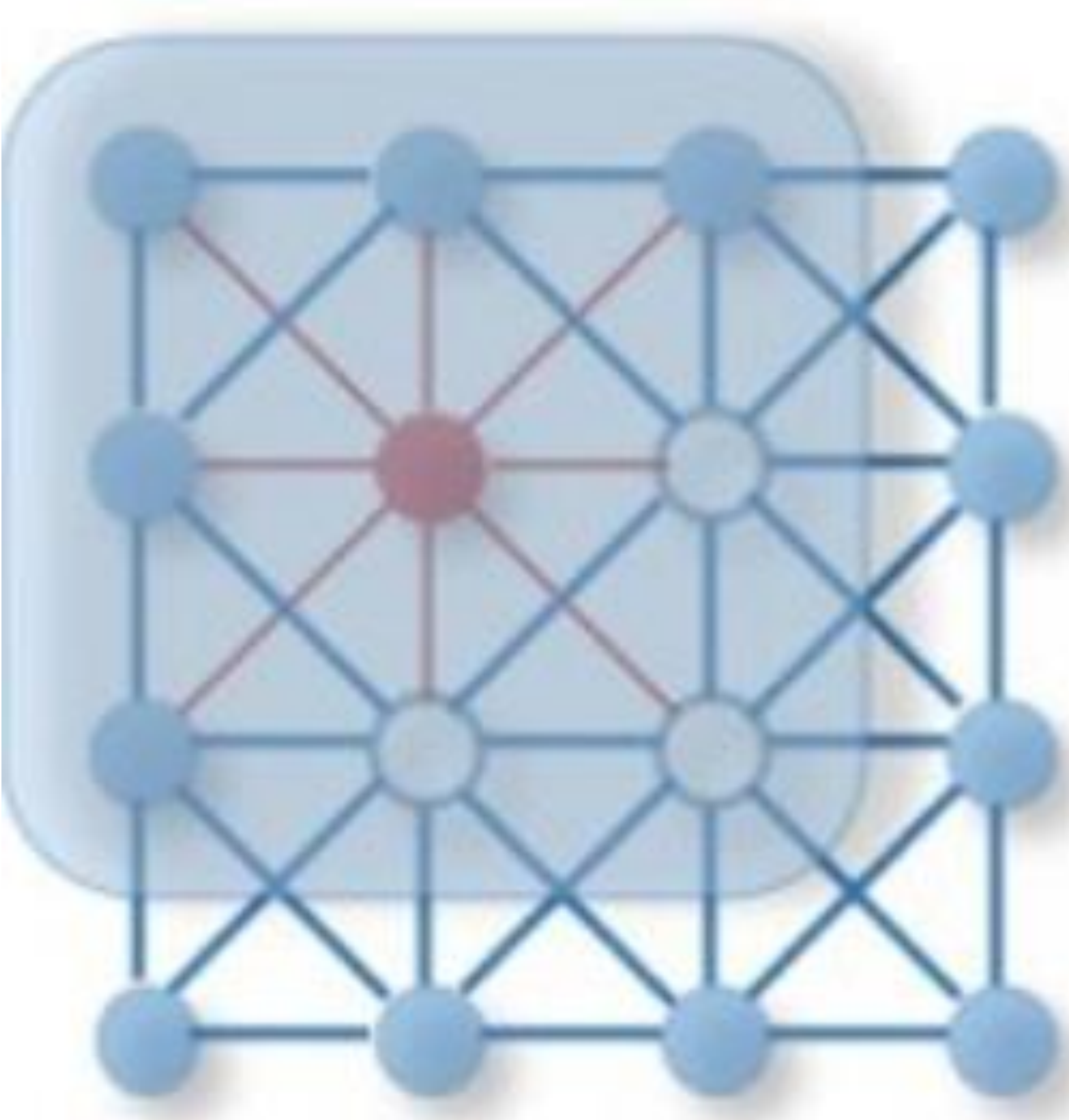


Image source
https://medium.com/@sunitachoudhary103/how-to-deal-the-graphs-data-in-deep-learning-with-graph-convolutional-networks-gcn-39f69db072ee

Convolution in graphs

- Derive a hidden representation of the red node, by taking the average value of the available features of the red node along with its neighbours

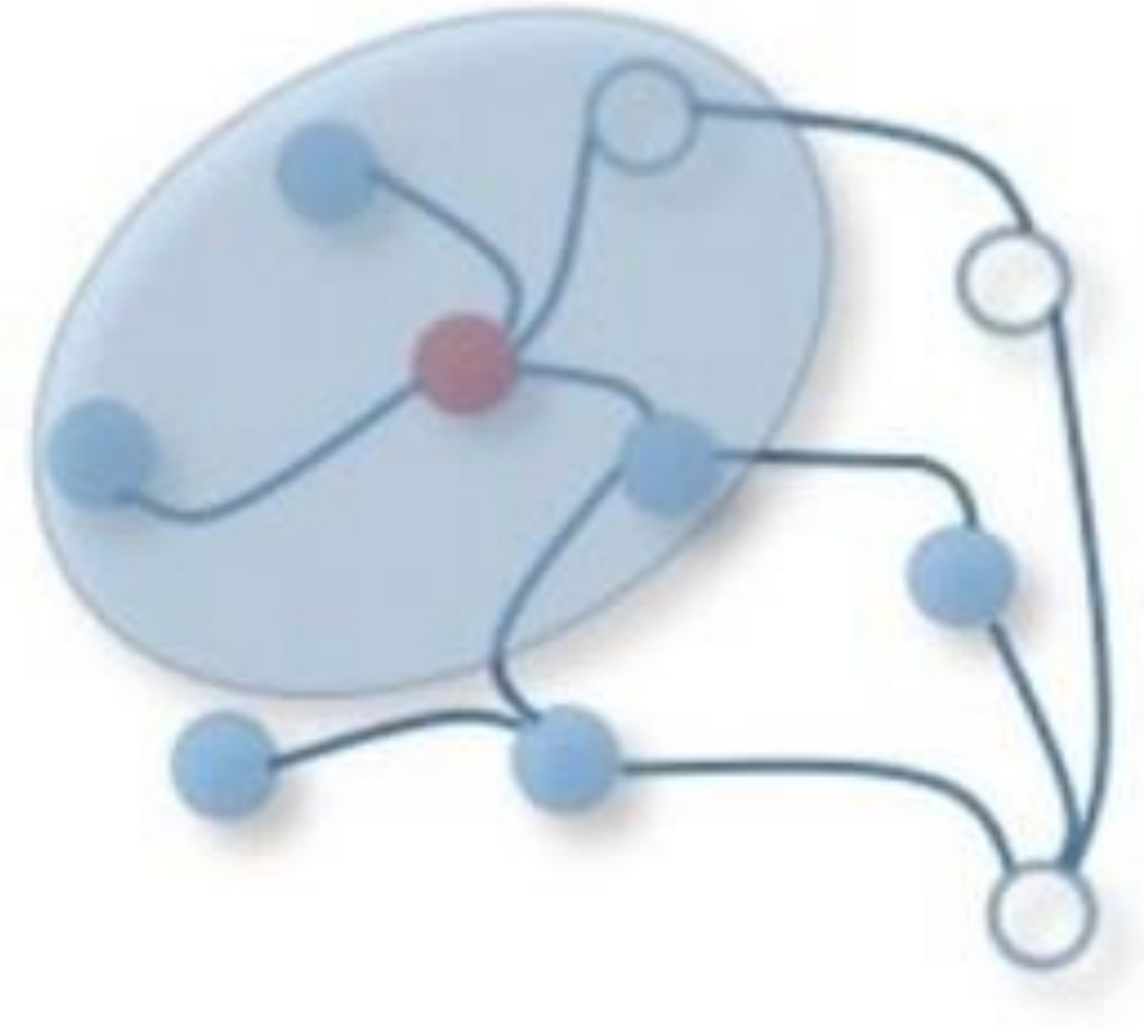
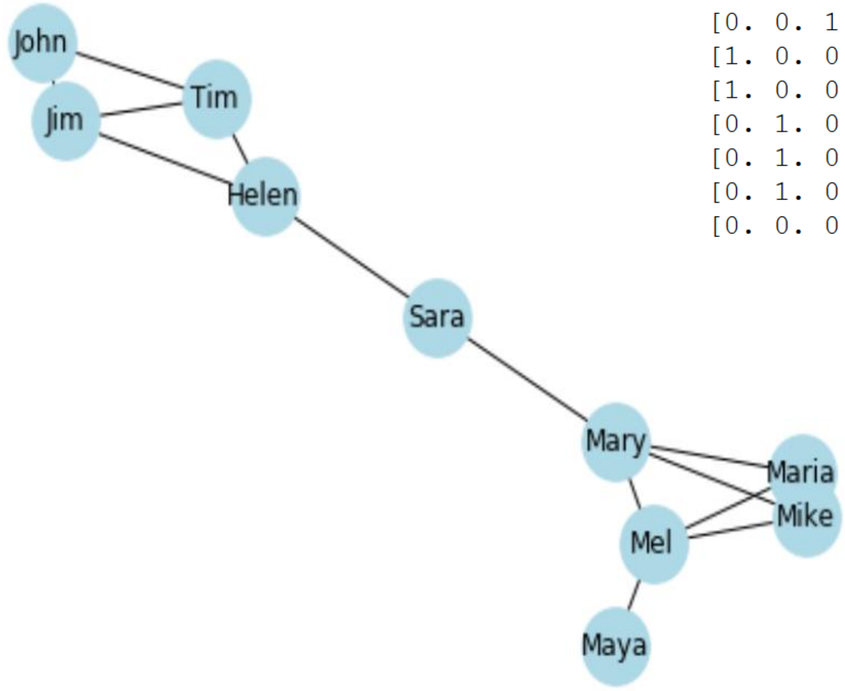


Image source
https://medium.com/@sunitachoudhary103/how-to-deal-the-graphs-data-in-deep-learning-with-graph-convolutional-networks-gcn-39f69db072ee

Let's see an example

Graph



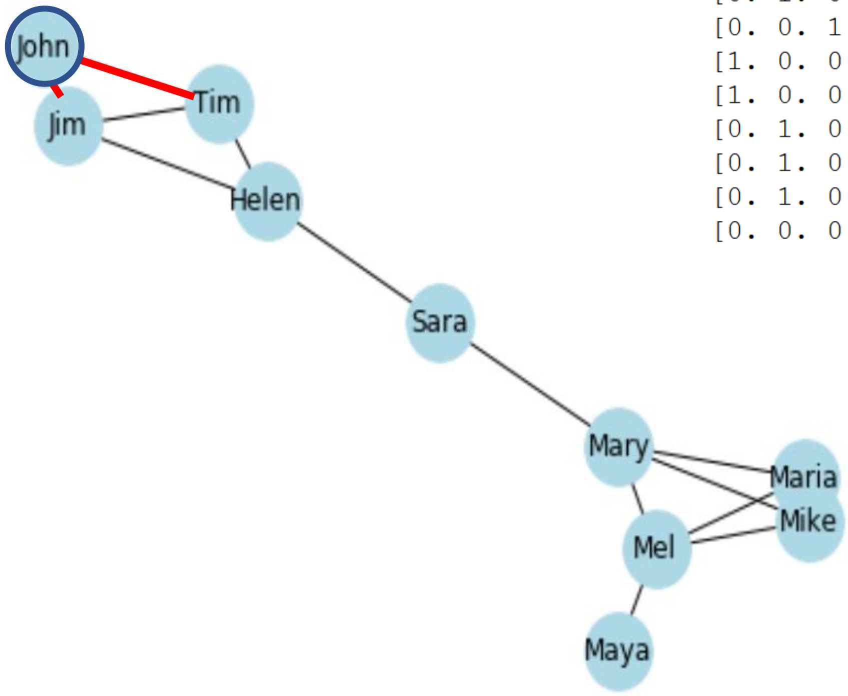

```
Node list:  ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
Original adjacency matrix:
 [[0. 0. 0. 0. 1. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 1. 1. 0.]
 [0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

Adjacency Matrix A

Note: in this example the graph is undirected, thus A is symmetric

Let's see an example

Graph



John →

```
Node list:  ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
Original adjacency matrix:
 [[0. 0. 0. 0. 1. 1. 0. 0. 0. 0.]
  [0. 0. 1. 0. 0. 0. 1. 1. 1. 0.]
  [0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]
  [0. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
  [1. 0. 0. 1. 0. 1. 0. 0. 0. 0.]
  [1. 0. 0. 1. 1. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
  [0. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
  [0. 1. 0. 0. 0. 0. 1. 1. 0. 1.]
  [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

Adjacency Matrix A

Note: in this example the graph is undirected, thus A is symmetric

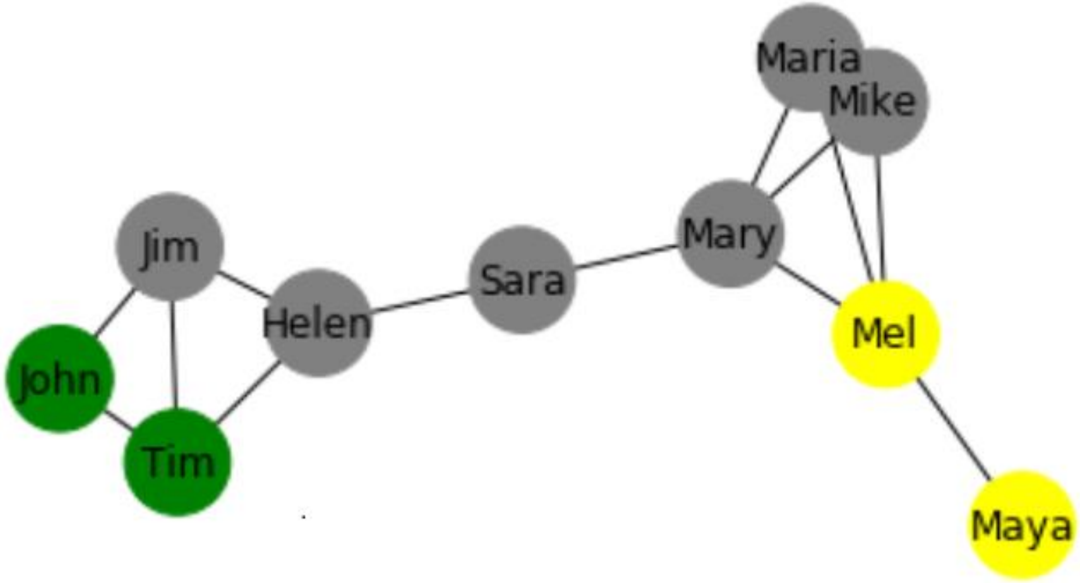
Encode two features using matrix X

(features don't have to be exclusive)

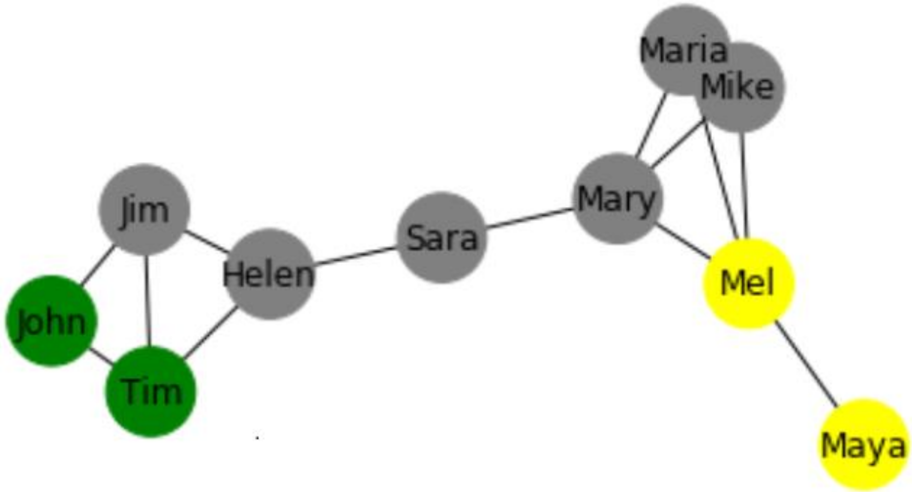
```
Node list:  ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
```

```
#PLAY WITH TWO FEATURES (PAO, AEK)
#John, Tim  = PAO
#Mel, Maya   = AEK

X=np.matrix([
    [1,0],
    [0,0],
    [0,0],
    [0,0],
    [1,0],
    [0,0],
    [0,0],
    [0,0],
    [0,1],
    [0,1]]
)
```



Aggregate features



- Let X be a n*k matrix encoding k features for each of the n nodes
- Question: what does **A*X** produce?

A

Mary →

```
[[0. 0. 0. 0. 1. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 1. 1. 0.]
 [0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

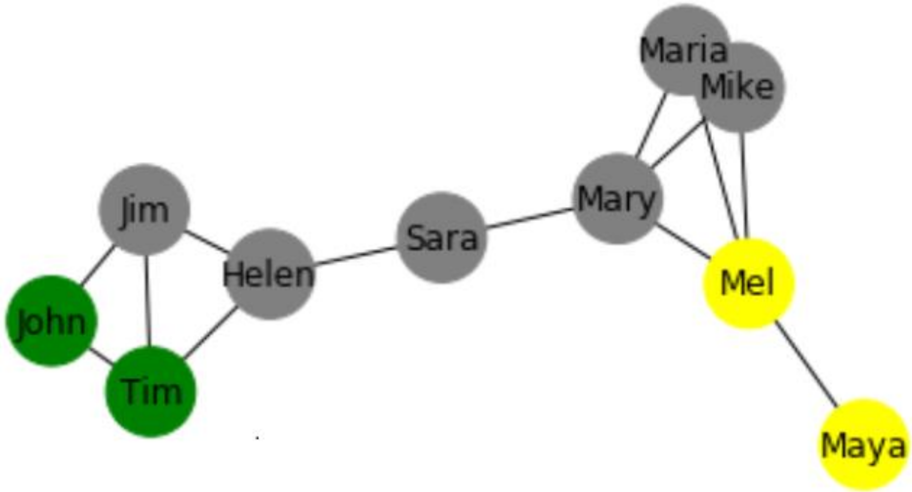
AEK

X

```
[[1 0]
 [0 0]
 [0 0]
 [0 0]
 [1 0]
 [0 0]
 [0 0]
 [0 0]
 [0 1]
 [0 1]]
```

```
Node list:  ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
```

Aggregate features



- Let X be a n*k matrix encoding k features for each of the n nodes
- Question: what does **A*X** produce?

A

Mary →

```
[[0. 0. 0. 0. 1. 1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 1. 1. 0.]
 [0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 1. 1. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 0. 1. 0.]
 [0. 1. 0. 0. 0. 0. 1. 1. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

AEK

X

```
[[1 0]
 [0 0]
 [0 0]
 [0 0]
 [1 0]
 [0 0]
 [0 0]
 [0 0]
 [0 1]
 [0 1]]
```

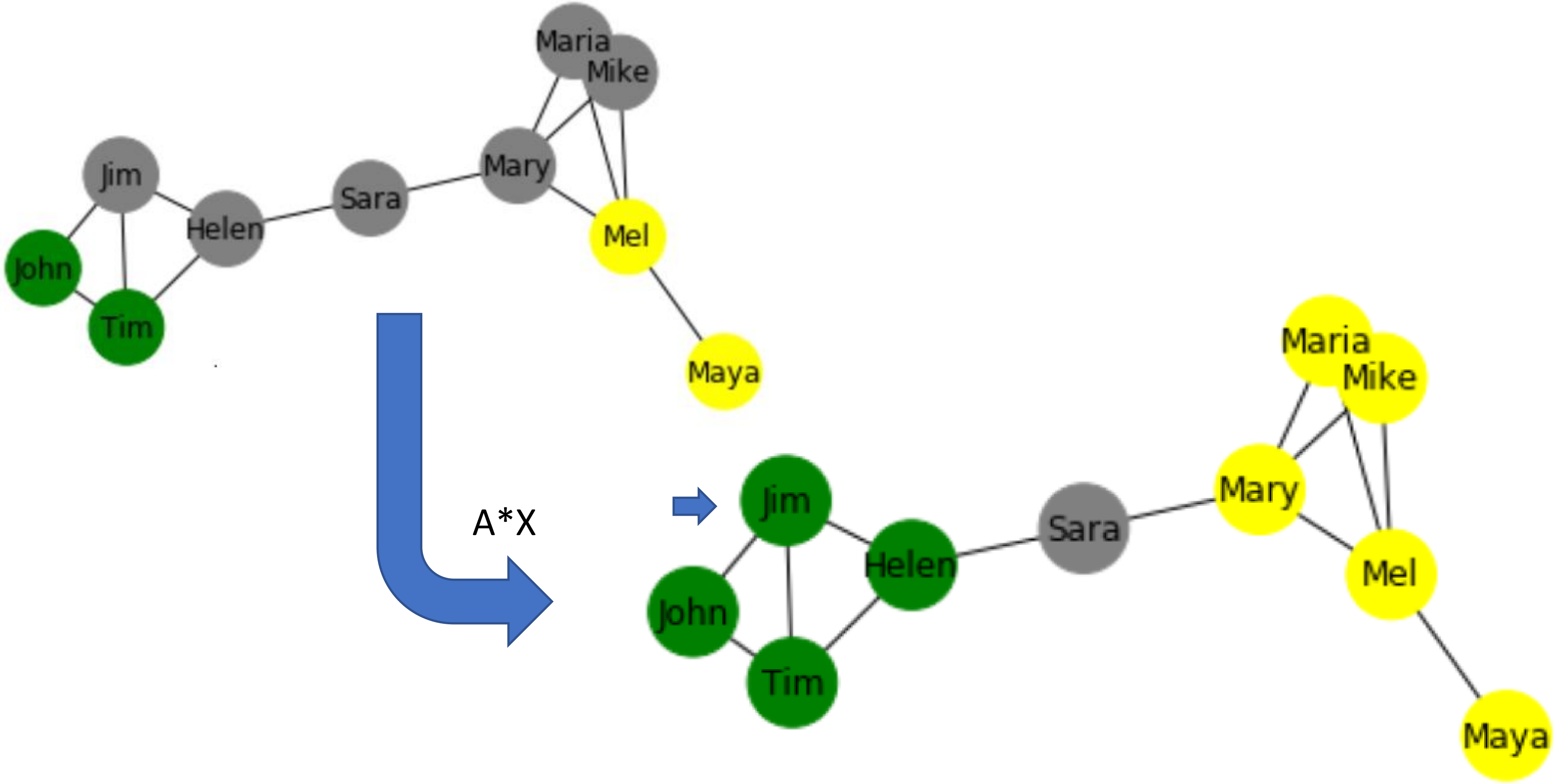
Thus, for each node we are aggregating the feature values of their neighbours!

```
Node list: ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
```

Result for our running example

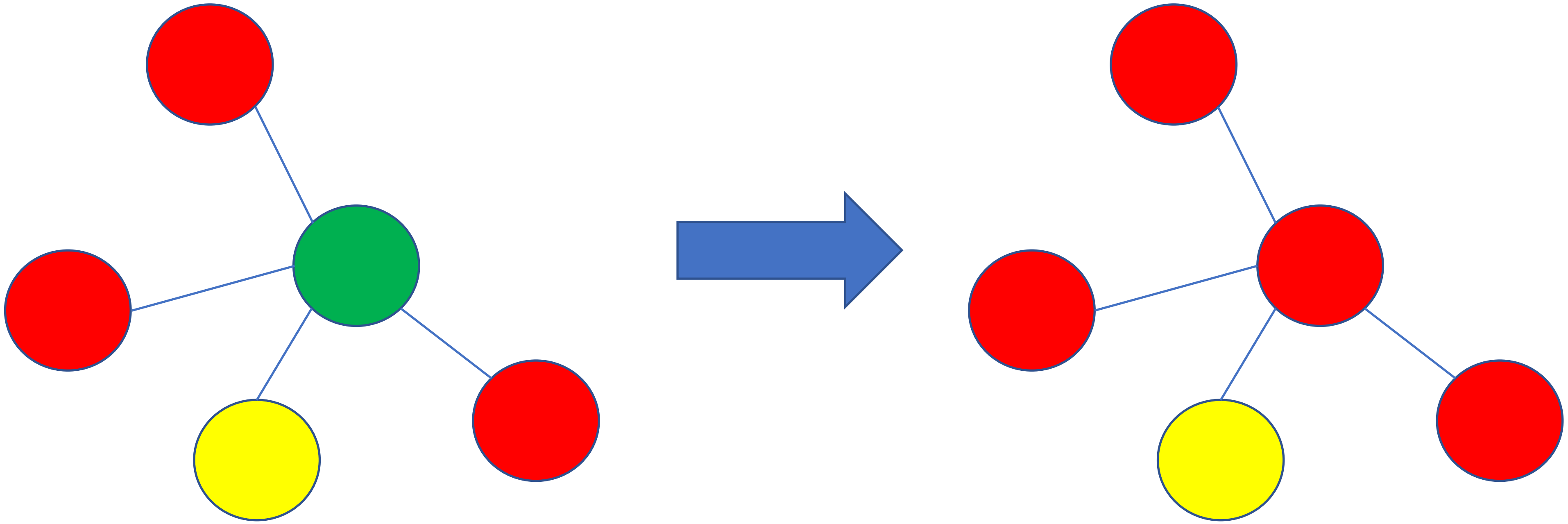
```
Node list:  ['John', 'Mary', 'Sara', 'Helen', 'Tim', 'Jim', 'Maria', 'Mike', 'Mel', 'Maya']
```

```
A*X=
 [[1. 0.]
  [0. 1.]
  [0. 0.]
  [1. 0.]
  [1. 0.]
  [2. 0.]
  [0. 1.]
  [0. 1.]
  [0. 1.]
  [0. 1.]]
```



Issue #1

- Node's own features are not taken into consideration in A*X
  - This is because A[i,i]=0



Issue #1: Solution

- Trick: add a self-loop
    - make A[i,i]=1
    - equivalently add identity matrix I: I[i,i]=1

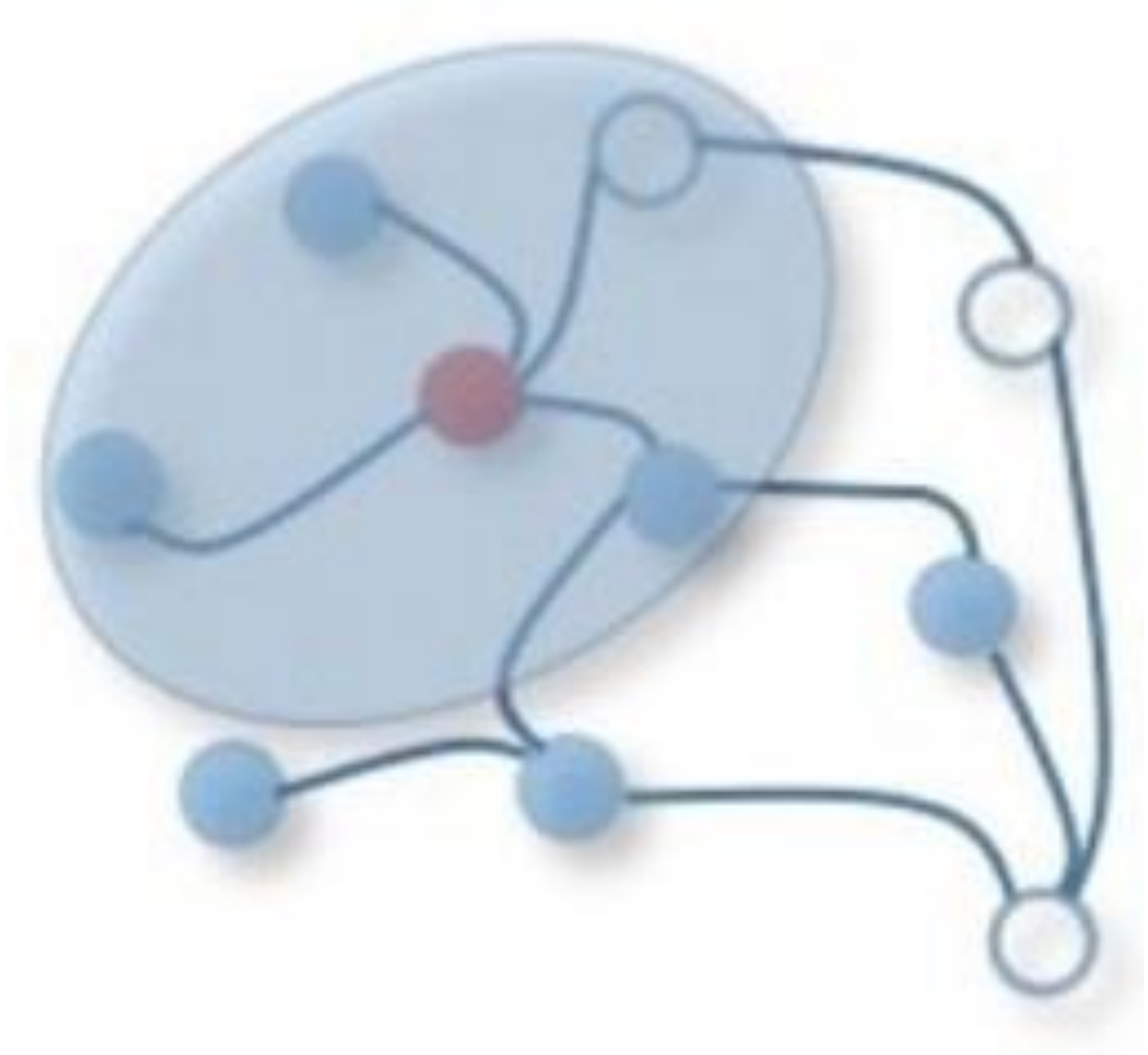
Issue #2

- A is not normalized. Thus, vertices with large degree will have large values in their feature representation while nodes with small degrees will have small values
  - Solve by using the symmetrically normalized adjacency matrix $D^{-0.5}(A + I)D^{-0.5}$

- D is a diagonal matrix with D[i,i] = degree of node i (computed on adjusted matrix A+I)
  - Lefthand side $D^{-0.5}$ scales the aggregate feature on i based on the degree on node i
  - Righthand side scales the aggregate feature on i based on the degree on node j
  - Intuition: often low-degree neighbours provide more useful information than high-degree neighbours

Recap (aggregation step)

- Compute normalized sum of neighboring nodes plus own features: $D^{-0.5}(A+I)D^{-0.5}\mathrm{X}$
- Where
  - A: Graph Adjacency matrix
  - I: Identity matrix
  - D: Degree matrix of A+I
  - X: Node's features



Graph Convolutional Networks

- In supervised learning we will use
  - $H^{(l+1)} = f(D^{-0.5}(A+I)D^{-0.5}H^{(l)}W^{(l)})$

  Where
  - $H^{(l)}$ is the input to layer l (initially the node features X we know from the dataset)
  - $H^{(l+1)}$ is the output to the next layer
  - $W^{(l)}$ are the weights to learn via training
  - f is an non-linear function such as ReLU

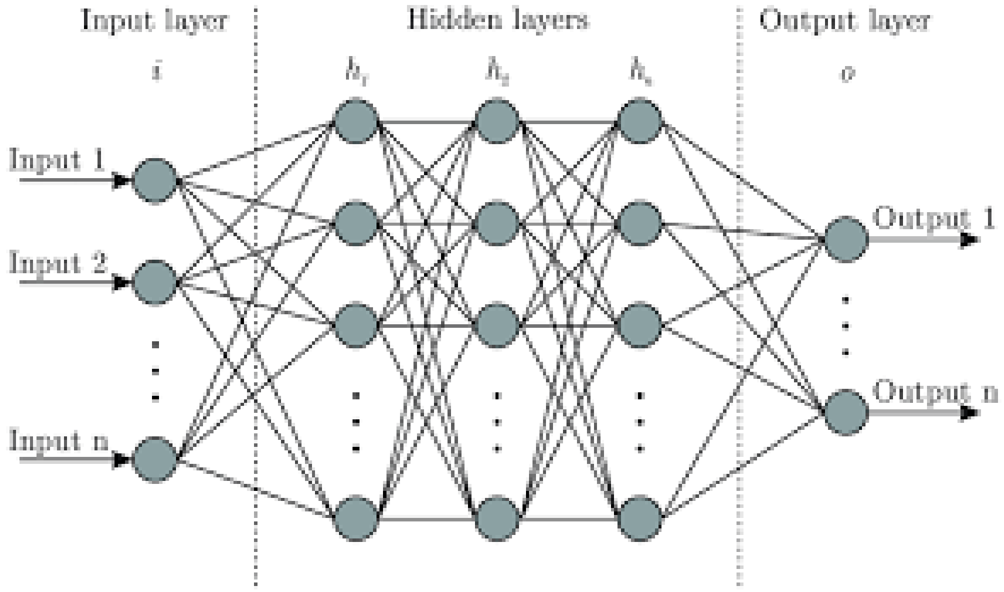


Image source: https://towardsdatascience.com/designing-your-neural-networks-a5e4617027ed

Continue our example

- Initialize nodes using I as X
- Use three hidden layers with random initial weights
- On the right see output with a single forward pass (no learning)