Graph Neural Networks Demystified

An overview of the essential concepts in Stanford CS224W (Lectures 1~8) with only oversimplified examples

Ng Yen Kaow

Embeddings

- Relatively small vectors associated with each object where similar objects have similar embeddings
- Using the embeddings of graph elements, various tasks can be performed
  - Cluster nodes in a graph
  - Predict properties of a node
  - Predict if two nodes may be connected
  - Classify entire graphs
- To perform each task, use the embedding with a suitable ML method
  - e.g. clustering can be performed with k-means

Obtaining embeddings

- Embeddings can be formed with or learned from features
  - Node-level features
    - Degree
    - Centrality (eigenvector/ betweenness/ closeness)
    - Clustering coefficient
    - Graphlets
    - Structure-based features
  - Link-level features
    - Distance-based features
    - Local/global neighborhood overlap
  - Graph-level features
    - Graph kernels
- Task-independent embeddings can be learned from unsupervised learning

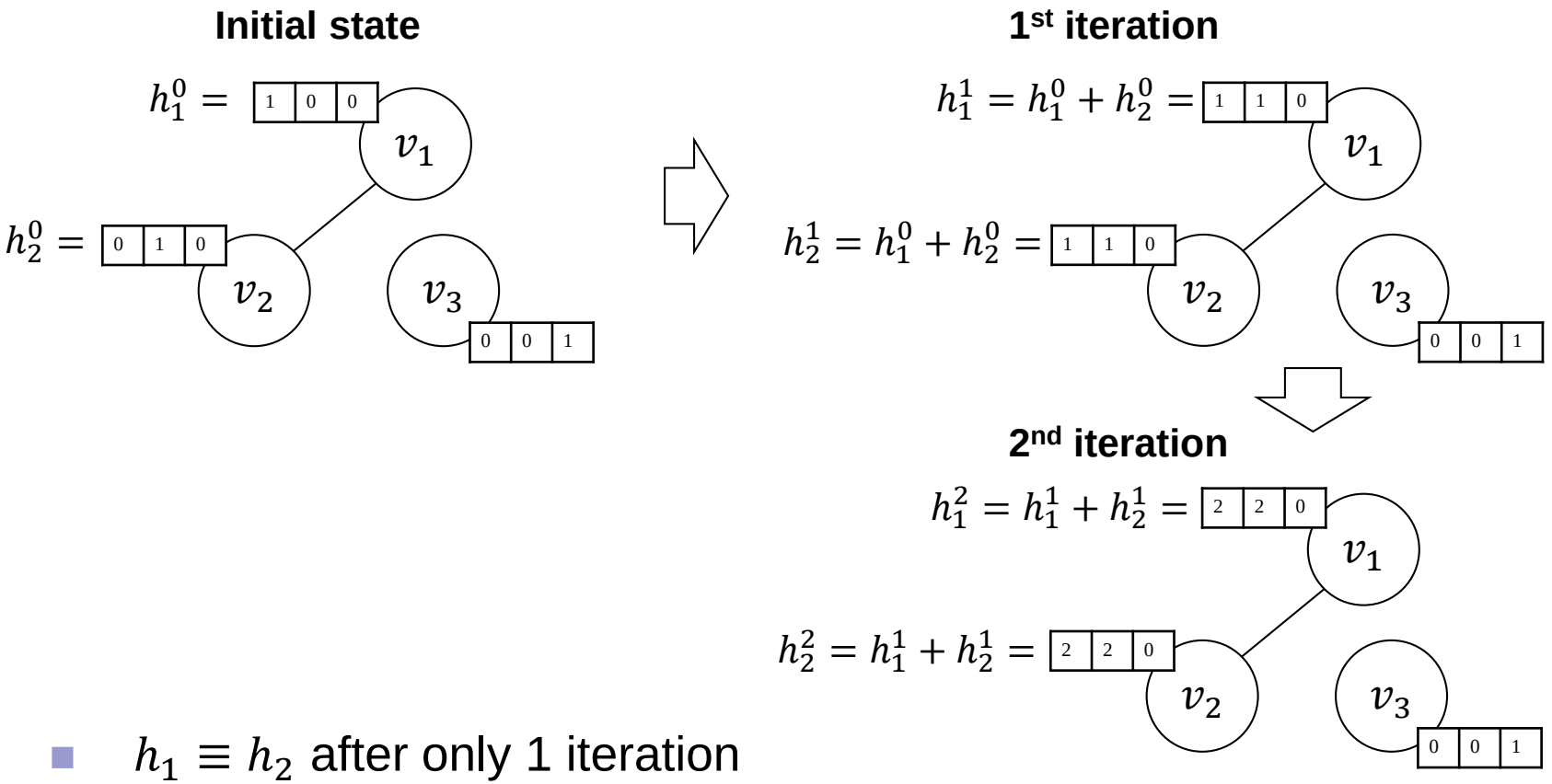
Task-independent embeddings

- Unsupervised extraction by random walks
  - **DeepWalk**
    - Estimate pairwise distance between nodes (hence their co-occurrence probability)
      - Usable for finding product relatedness in recommender
    - Node embeddings
      1. Estimate node distances with random walks
      2. Train a neural network (with node input and embedding output) such that distances between embeddings agree with estimated distances
  - **Anonymous Walk**
    - Embeddings for entire graphs
  - **Simpler method: just add up neighbors**

Embeddings by adding neighbors

- Sum up the features of (self and) neighbor nodes
  - Features of nodes in close proximity will become similar

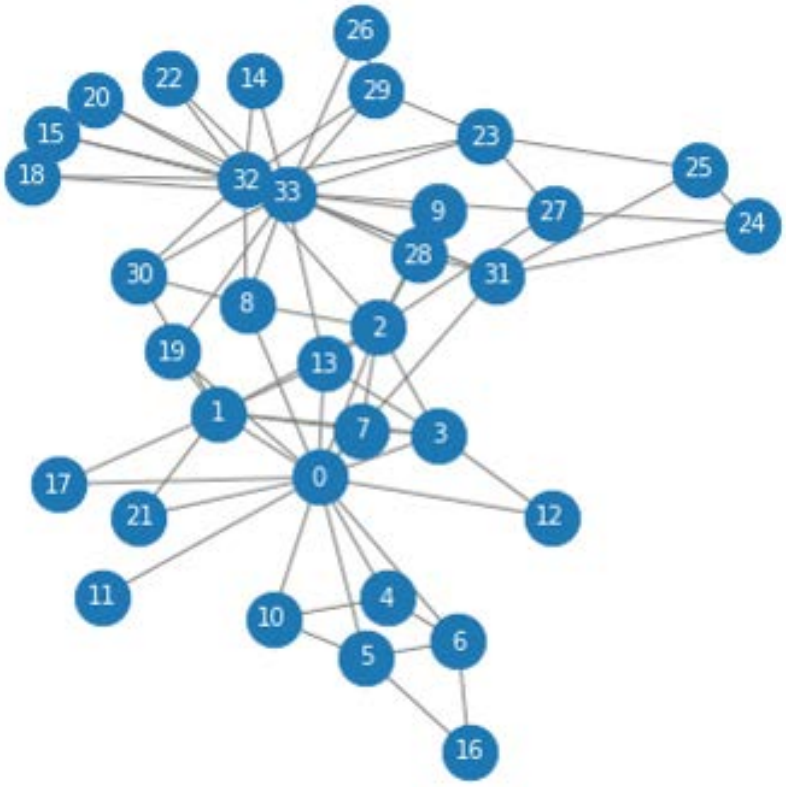
Example: Let h_i^j denote features of node i at iteration j and let $h_1^0 = (1 \quad 0 \quad 0)$, $h_2^0 = (0 \quad 1 \quad 0)$, and $h_3^0 = (0 \quad 0 \quad 1)$



- $h_1 \equiv h_2$ after only 1 iteration

Embeddings by adding neighbors

- To cluster nodes in a graph, will it work if we
  1. Start with a unique feature for each node, and
  2. Repeatedly add up neighboring features, and
  3. Finally, cluster the resultant features with some method like k-means?



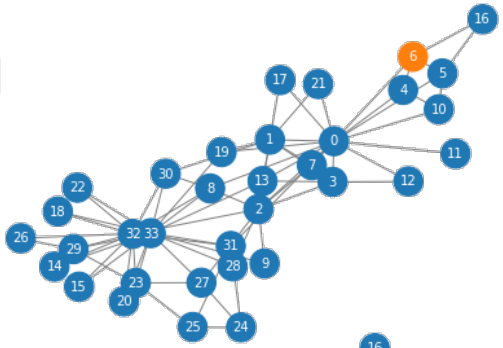
Let's try with karate club network

Based on https://tkipf.github.io/graph-convolutional-networks/

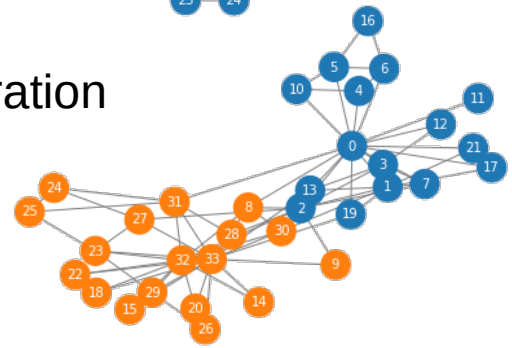
Embeddings by adding neighbors

- Karate club

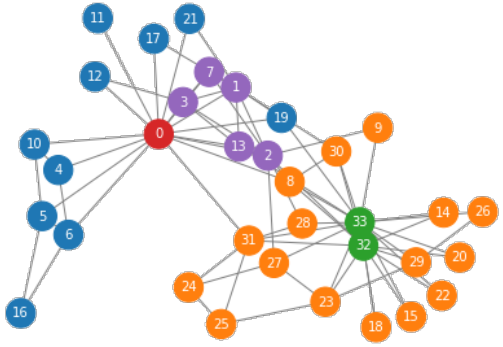
Initial



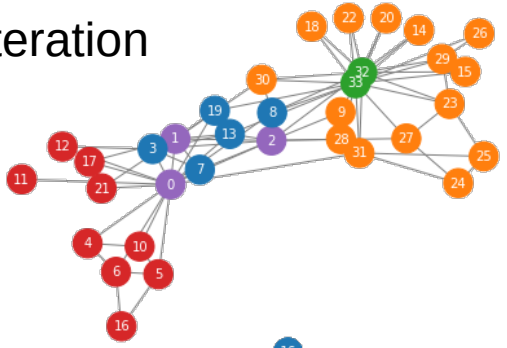
1st iteration



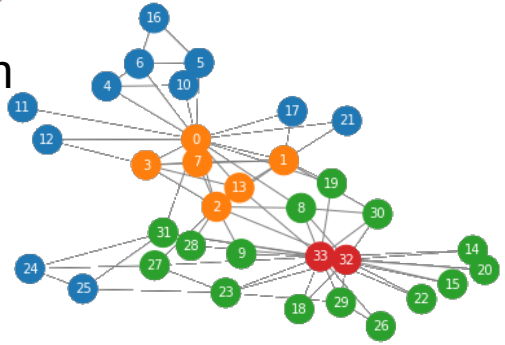
2nd iteration



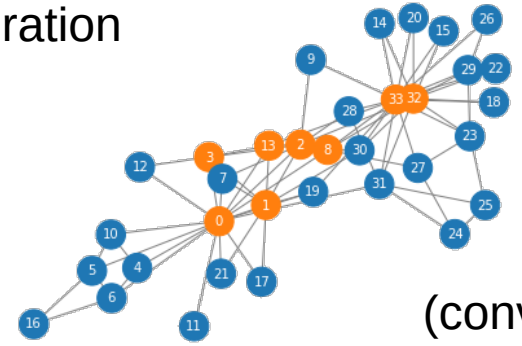
3rd iteration



4th iteration



5th iteration



(converged)

(number of clusters determined with silhouette method)

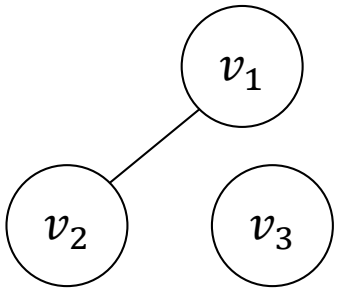
Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Then, **sum** is simply $\hat{A}H$

$$\begin{pmatrix} a & b & c \\ & \dots & \\ & \dots & \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} ah_1 + bh_2 + ch_3 \\ \dots \\ \dots \end{pmatrix}$$

Permutation invariant so that the outcome is the same regardless of node order within matrix

e.g.



$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} h_1 + h_2 \\ h_1 + h_2 \\ h_3 \end{pmatrix}$$

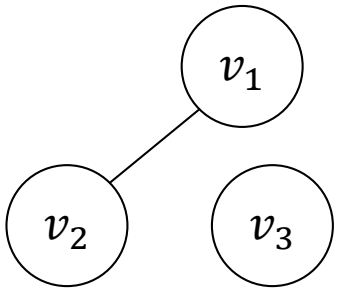
Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Further **normalize** each row of $\hat{A}$ to sum to 1

$$\begin{pmatrix} 1/3 & 1/3 & 1/3 \\ \dots & & \\ \dots & & \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} (h_1 + h_2 + h_3)/3 \\ \dots \\ \dots \end{pmatrix}$$

Note that **normalize** does the same thing as **mean**

e.g.



$$\begin{pmatrix} .5 & .5 & 0 \\ .5 & .5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} (h_1+h_2)/2 \\ (h_1+h_2)/2 \\ h_3 \end{pmatrix}$$

Adding neighbors w/ linear algebra

- Let matrix H be a matrix where each row is a node and each column is a feature
  - H have dim $|V| \times d$
- Let A be an adjacency matrix
  - Let $\hat{A} = A + I$ where I is the identify matrix
- Further **normalize** each row of $\hat{A}$ to sum to 1
  - To perform this normalization, it suffices that we let $\hat{A} \leftarrow D^{-1}\hat{A}$ where D is the diagonal node degree matrix
    - In PyTorch, use

      ```
      torch.nn.functional.normalize(A, p=1, dim=1)
      ```

  - Or, use $\hat{A} \leftarrow D^{-1/2}\hat{A}D^{-1/2}$ (GCN variant)
    - In PyTorch, use

      ```
      D = torch.diag(torch.sum(A, 1)).inverse().sqrt()
      D = torch.mm(torch.mm(D, A), D)
      ```

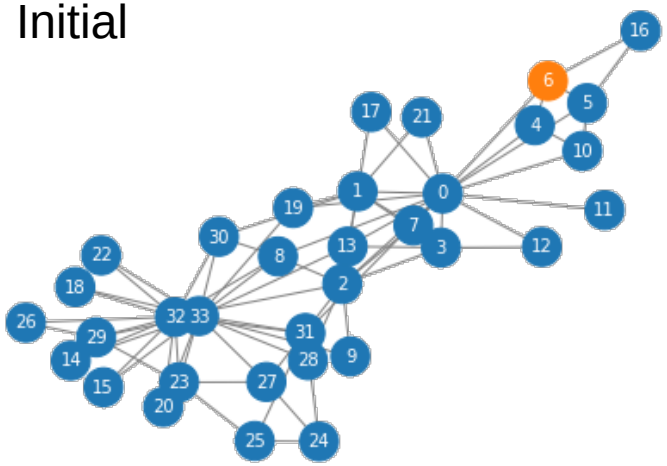
Normalized $\hat{A}$ is in general **not symmetric**

GCN variant is symmetric, but **not normalized**

Redo embeddings w/ $D^{-1}\hat{A}$

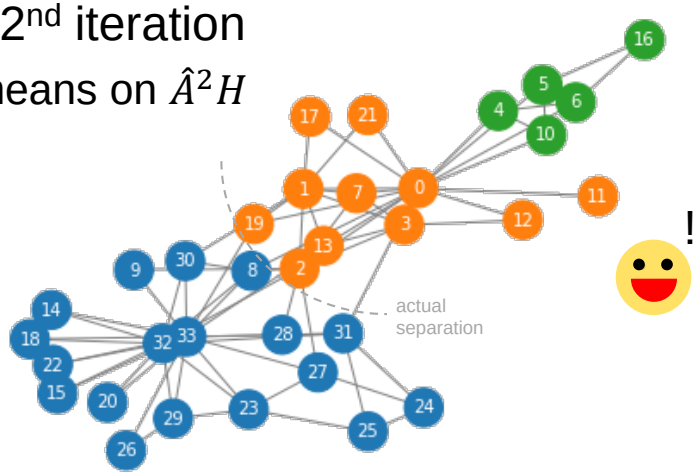
- Redo karate club with normalized $\hat{A} \leftarrow D^{-1}\hat{A}$

Initial



2nd iteration

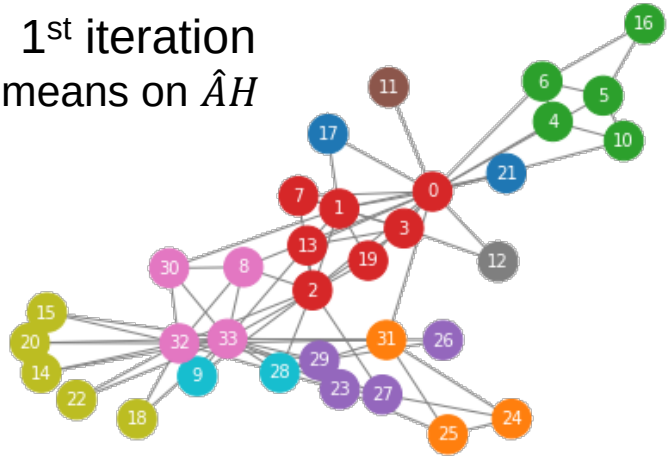
k-means on $\hat{A}^2H$



!

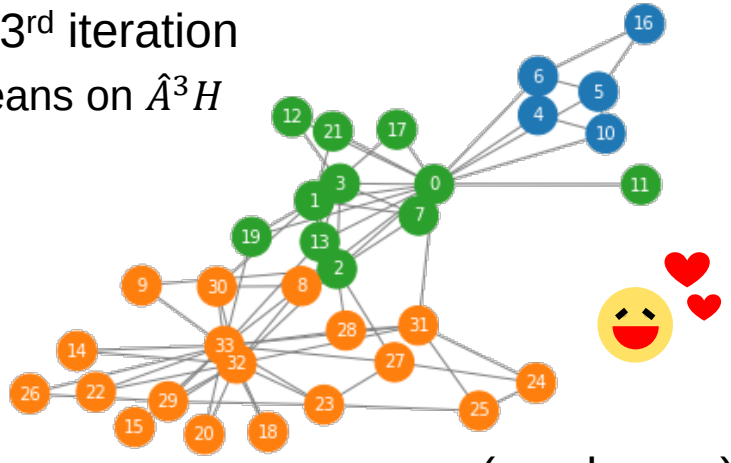
1st iteration

k-means on $\hat{A}H$



3rd iteration

k-means on $\hat{A}^3H$



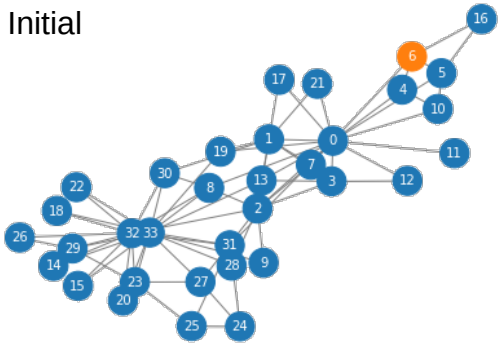
(no change)

(number of clusters determined with silhouette method)

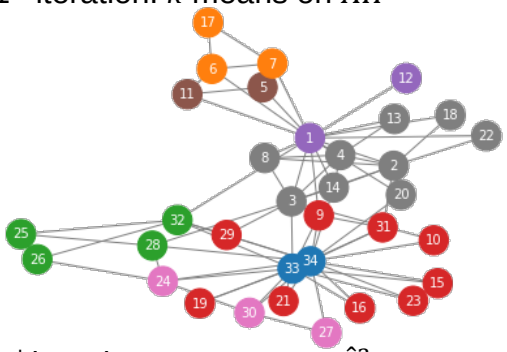
Redo embeddings w/ $D^{-1/2}\hat{A}D^{-1/2}$

- Redo karate club with normalized $\hat{A} \leftarrow D^{-1/2}\hat{A}D^{-1/2}$

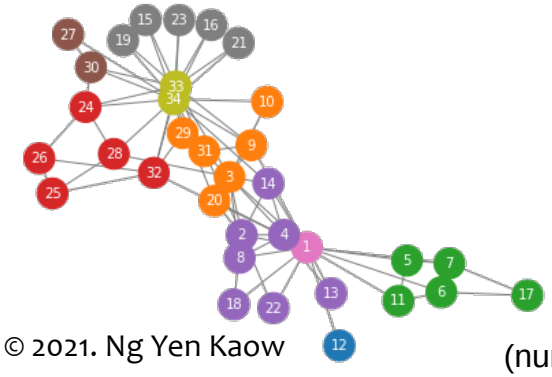
Initial



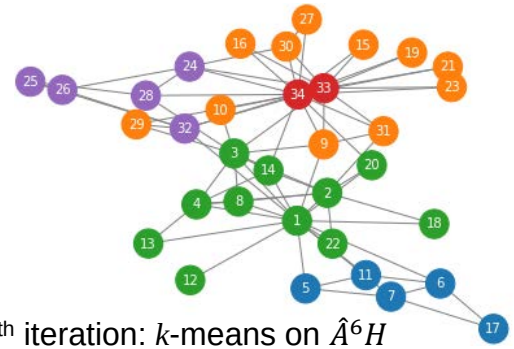
1st iteration: k-means on $\hat{A}H$



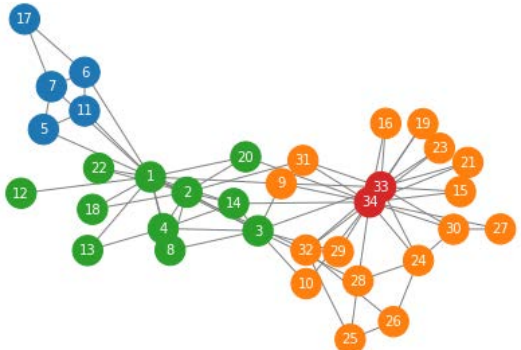
2nd iteration: k-means on $\hat{A}^2H$



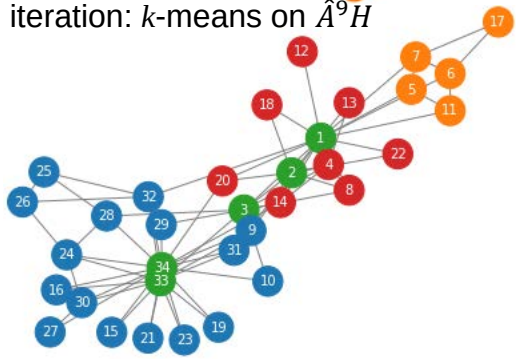
3th iteration: k-means on $\hat{A}^3H$



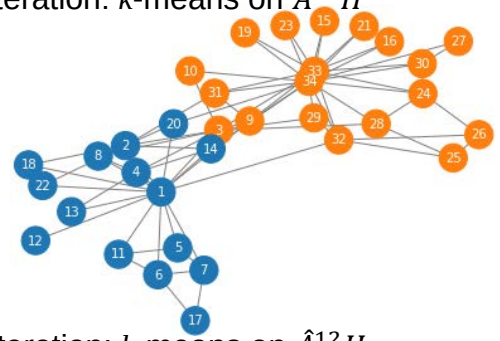
6th iteration: k-means on $\hat{A}^6H$



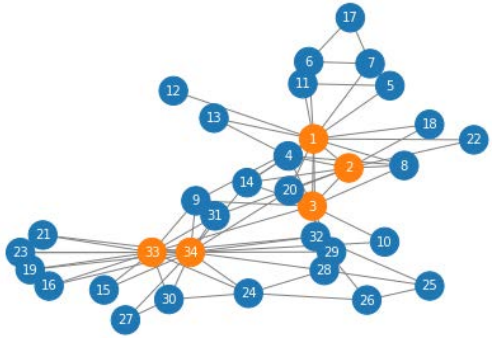
9th iteration: k-means on $\hat{A}^9H$



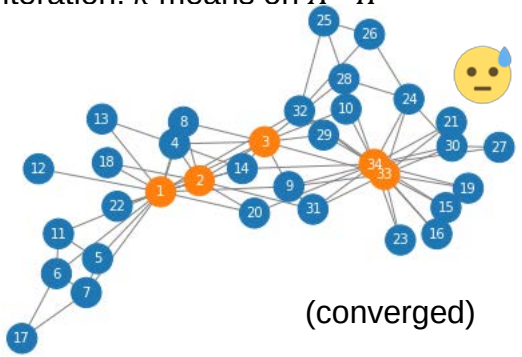
11th iteration: k-means on $\hat{A}^{11}H$



12th iteration: k-means on $\hat{A}^{12}H$



13th iteration: k-means on $\hat{A}^{13}H$



(converged)

(number of clusters determined with silhouette method)

Adding neighbors: evaluation

- Why did it fail without normalization
  - Without normalization, feature values for the nodes of high centrality would quickly add up, making them distinct from the nodes of low centrality
- Why did it fail with $D^{-1/2}\hat{A}D^{-1/2}$ (GCN variant)
  - This is more complicated and is explained in the slide titled $\hat{A}H^* = H^*$ for symmetric $\hat{A}$
- How many iterations should be used?

Early (RNN-like) GNNs are iterated until convergence but they quickly ran out of favor to Graph Convolutional Networks (GCNs) where the number of iterations is fixed as defined by the number of convolutional layers

  - Each iteration would "bunch up" neighboring features of 1 hop away (receptive field)
  - We should determine the number of iterations by the **nature of the graph**

Nature of the graph

- The **Cheeger constant** (or expansion constant) of an **unweighted** graph $G(E, V)$ is

$$h(G) = \min_{S \subseteq V} \frac{|\{(u,v)|u \in S, v \in \bar{S}\}|}{\min(|S|, |\bar{S}|)}$$

  - $|\{(u, v)|u \in S, v \in \bar{S}\}|$ indicates how well vertices in S are connected to vertices in $\bar{S}$
  - $\min(|S|, |\bar{S}|)$ favors S where $|S| \approx |\bar{S}|$
  - For weighted graphs, a similar measure called conductance can be defined with edge weights (a_{uv})

$$\phi(G) = \min_{S \subseteq V} \frac{\sum_{v \in S, u \in \bar{S}} a_{vu}}{\min\left(\sum_{v \in S, u \in V} a_{vu}, \sum_{v \in \bar{S}, u \in V} a_{vu}\right)}$$

Based on https://dspace.mit.edu/handle/1721.1/121660

Not in CS224W

Nature of the graph

- The **Cheeger constant** (or expansion constant) of an **unweighted** graph $G(E,V)$ is

$$h(G) = \min_{S \subseteq V} \frac{|\{(u,v)|u \in S, v \in \bar{S}\}|}{\min(|S|,|\bar{S}|)}$$

- A large $h(G)$ indicates a **highly-connected graph**
  - A feature in a highly-connected graph will propagate in the graph very quickly
  - A random walk in a highly-connected graph converges in $O(\log|V|)$ steps to an almost uniform distribution (mixing time)
    - Upon which the embedding of every node is influenced almost equally by any other node

Nature of the graph

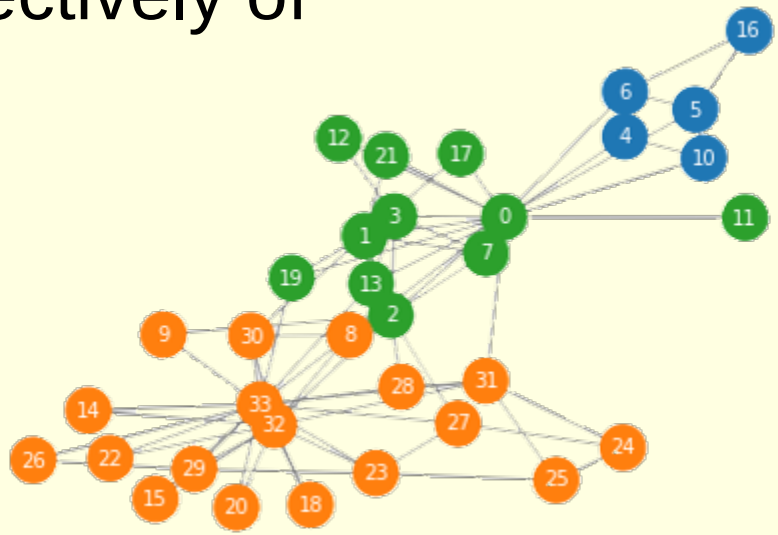
- Examine the number of steps required for the clusters in the karate club to mix

- The clusters are respectively of sizes 18, 11, 5

$$\log(18) = 4.17$$

$$\log(11) = 3.46$$

$$\log(5) = 2.32$$

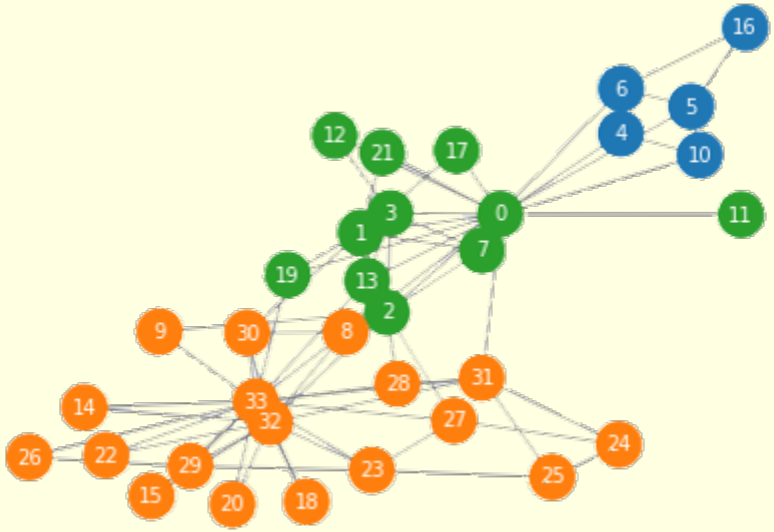


- 5 iterations/ steps suffice for nodes in the respective clusters to influence each other equally

Nature of the graph

- Examine the number of steps required for the clusters in the karate club to mix

- Will increasing the number of iterations eventually spread the features uniformly across the entire karate club graph?
  - Depends on whether the limiting distribution H^* (i.e. when $\hat{A}H^* = H^*$) is everywhere constant (next slide)



$\hat{A}H^* = H^*$ for symmetric $\hat{A}$

- If $\hat{A}$ is symmetric (and hence can be eigendecomposed), then each application of $\hat{A}$ on H, $\hat{A}H = U\Lambda U^\top H$
  - This is elaborated in the slides on the spectral basis of GNN
- For 2 layers, $\hat{A}^2 H = U\Lambda U^\top (U\Lambda U^\top H) = U\Lambda^2 U^\top H$
  $\Rightarrow$ For k layers, $\hat{A}^k H = U\Lambda^k U^\top H$
  - λ^k of larger λ becomes disproportionately large
  - At large k, $\hat{A}^k H$ **is a projection of H mainly on the eigenvectors of the largest eigenvalues**
- **For the adjacency matrix $\hat{A}$ (or A), a larger eigenvalue implies more similar values in its eigenvector**
  - Note that the Laplacian $(D - A)$ or the normalized Laplacian $(I - D^{-1/2}AD^{-1/2})$ reverses this relation
- As a result, $\hat{A}^k H$ **consist of similar features**, leading to most everything clustered together

Compared to spectral clustering

We compare $\widehat{D}^{-1/2}\hat{A}\widehat{D}^{-1/2}$ with $D^{-1/2}AD^{-1/2}$ (Ng, Weiss, and Jordan 2001) since they share more similarity

- Spectral clustering finds the distribution x where

$$D^{-1/2}AD^{-1/2}x = x$$

That is, x is the eigenvector of eigenvalue 1

- For **single-valued feature** ($H \leftarrow x$) and **at convergence** ($\hat{A}x = x$), our earlier GNN gives x where

$$\widehat{D}^{-1/2}\hat{A}\widehat{D}^{-1/2}x = x$$

where $\hat{A} = A + I$ and $\widehat{D}$ the corresponding degree matrix

  - The resultant features, $\hat{A}^{\infty}x$, are dominated by the eigenvectors of the largest eigenvalues

- Self-loops in $\hat{A}$ shrink the spectrum of the Laplacian

⇒ Faster domination by the larger eigenvalues (see slides for “Spectral Basis of GNN”)

Adding neighbors: evaluation

- Benefits of strategy
  - Simplicity
  - Efficiently computed with adjacency matrix
- Disadvantage of strategy
  - Embeddings produced are of size of the number of nodes in the graph

    ⇒ Learn a **transformation matrix**

    $W: R^{|V|} \rightarrow R^{d}$ for some smaller d

Transformation matrix W

- W is typically a linear transformation layer of size $|V| \times d$ where d is the target dimensionality of the embeddings
- Combined with the adjacency matrix $\hat{A}$, we now have a complete matrix formulation for computing embedding h_v of a node v from (itself and) its neighbors, in the form of

$$h_v \leftarrow \left(\hat{A}\right)_v HW$$

where

  - $\left(\hat{A}\right)_v$ is the row in $\hat{A}$ for the node v, and
  - H is a matrix containing the features/embeddings of all the nodes (of course, only the rows in H with non-zero entries in $\left(\hat{A}\right)_v$ are needed for computing h_v)

- **Variations in this formula lead to various frameworks**

Variations

- Message-aggregation (MSG-AGG)
  - First transform features/embeddings (MSG), then aggregate transformed embeddings (AGG)

$$h_v \leftarrow \underbrace{\left(\hat{A}\right)_v}_{\text{aggregate}} \overbrace{(HW)}^{\text{message}}$$

- Separate computation of self and neighbors
  - Exclude entry for v from $\left(\hat{A}\right)_v$, and let

$$h_v \leftarrow \mathrm{AGG}\left(\overbrace{\left(\hat{A}\right)_v HW}^{\text{Aggregate only neighbors}}, \overbrace{h_v W'}^{\text{Self}}\right)$$

Learn a different transformation for self

Also denoted as B

where AGG is, for instance, concatenation

Frameworks

- **Graph Convolutional Network** (GCN)

  $h_v \leftarrow \left(\hat{A}\right)_v (HW)$ (basically just MSG-AGG)

  (See Graph Fourier Transform in later slides to understand the significance of this simple framework)

- **GraphSAGE**
  - Exclude entry for v from $\left(\hat{A}\right)_v$

$$h_v \leftarrow \left(\text{CONCAT}\left(\text{AGG}\left(\left(\hat{A}\right)_v H\right), h_v\right)\right) W$$

Aggregate neighbors; Self; Concatenate self & aggregated neighbors; Transform

AGG can be one of many options including MLP, LSTM, *etc.*

(Why use these? See Graph Isomorphism Network)

⇒ AGG is learnable

Frameworks

- **Graph Attention Networks** (GAN)
  - Instead of learning AGG, learn $\hat{A}$
    - Generalize the adjacency matrix $\hat{A}$ to **attention weights** $\Lambda = (\alpha_{vu})$

$$h_v \leftarrow \left(\hat{A}\right)_v HW \Rightarrow h_v \leftarrow (\Lambda)_v HW$$

where $\alpha_{vu} = \dfrac{\exp(e_{vu})}{\sum_{x \in N(v)} \exp(e_{vx})}$, and

e_{vu} is a measure of how related u and v are

- e_{vu} is usually computed as $\mathrm{LINEAR}\left(\mathrm{CONCAT}(h_v W, h_u W)\right)$

  - Do not confuse with Generative Adversarial Networks which is for generating anime pics

- Implemented in PyTorch Geometric (PyG) as GCNConv (GCN), SAGEConv (GraphSAGE), and GATConv (GAN)
  - See https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

Frameworks

- **Message Passing Neural Network** (MPNN)
  - Involve $N(v)$ in the transformation W for v

    $$h_v \leftarrow \left(\hat{A}\right)_v HW$$

    $$\Rightarrow h_v \leftarrow H \bigoplus_{u \in N(v)} \phi(h_v, h_u)$$

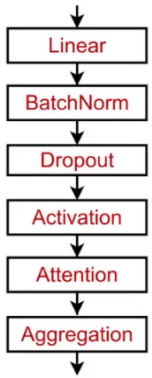
    This change allows us to incorporate edge features in the embedding

    $$\Rightarrow h_v \leftarrow H \bigoplus_{u \in N(v)} \phi(h_v, h_u, e_{vu})$$

  - How to compute $\phi(h_v, h_u, e_{vu})$ algebraically?
    - Let edge features be in a 3D matrix E
    - Then, $\left(\hat{A}\right)_v H$ and $\left(\hat{A}\right)_v (E)_v$ gives us two matrices with matching rows (each row corresponding to h_u and e_{vu} respectively)
    - Concatenate $\left(\hat{A}\right)_v H$ and $\left(\hat{A}\right)_v (E)_v$ and give as input to an NN
- A similar framework, Principal Neighborhood Aggregation (PNAConv), is implemented in PyG (these frameworks are not discussed in CS224W)

In practical use

- At this point we have not mentioned activation function or other elements of DL
  - For activation function just let $h_v \leftarrow \sigma(h_v)$
  - Mix and match as you like



- Embeddings can be used for many downstream tasks
  - We have earlier used k-means for clustering the final output
  - Better performed by constructing a neural network directly with the GNN layers

In practical use

- Adding graph elements
  - Features
    - Similar to feature engineering
  - Virtual nodes
    - Connecting all the nodes in a sparse but apparent subgraph to a virtual node will allow those nodes to better communicate
  - Virtual edges
    - Create new graph by systematically adding edges
    - Example: Given a bipartite graph, breaking the graph into two of only nodes of the same type is good for some analyses
      - Let A be the adjacency matrix of the bipartite graph G
      - A^2 then gives the number of paths of distance 2 between nodes in G
        ⇒ an adjacency matrix between nodes of the same type
        ⇒ allows us to separate G into two graphs, each of same node type
      - $A + A^2$ can form an adjacency matrix with heterogeneous edges

Training GNNs

- Using node embeddings as input to a prediction function
  - Embedding of 1 node can be used directly
  - Embeddings of 2 nodes can be
    - Concatenated to form an **edge embedding**
    - Projected on each other to get their similarity
  - Embeddings of nodes of the entire graph can be
    - Summed, averaged, searched for max/min, *etc.*
    - Clustered, then the clusters summed, average, *etc.*, in a hierarchical fashion
- **Edge embeddings from edge features** are also possible, though not discussed in CS224W
  - The framework **Node and Edge features in graph Neural Networks** (NENN) (not yet in PyG)