50.039 Theory and Practice of Deep Learning

W9-S2 Graph Convolutional Networks

Matthieu De Mari

About this week (Week 9)

1. What are **graph objects**?
2. How do we **define** a graph object **mathematically**?
3. What are **typical graph problems**?
4. How can we **embed a graph object** to later feed it to a Neural Network?
5. What is a **graph convolution** and how des it relate to the concept of image convolution?
6. What are more **advanced problems** and **approaches** on graph convolutional Neural Networks?

Outline

In this lecture

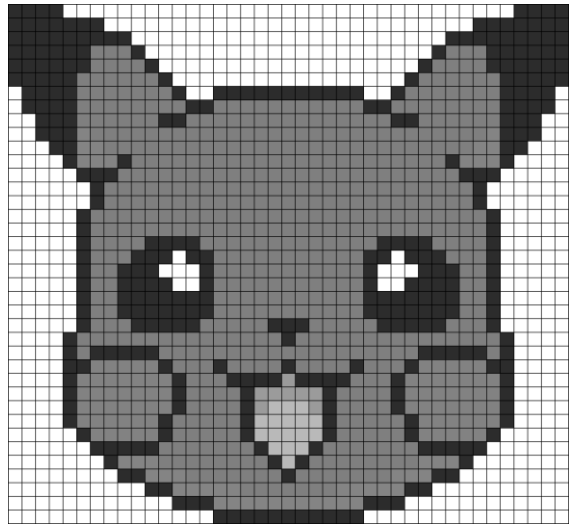
- Using graph types datasets
- Graph convolutions and graph embeddings
- How these convolutions relate to images
- Graph Convolutional Neural Networks

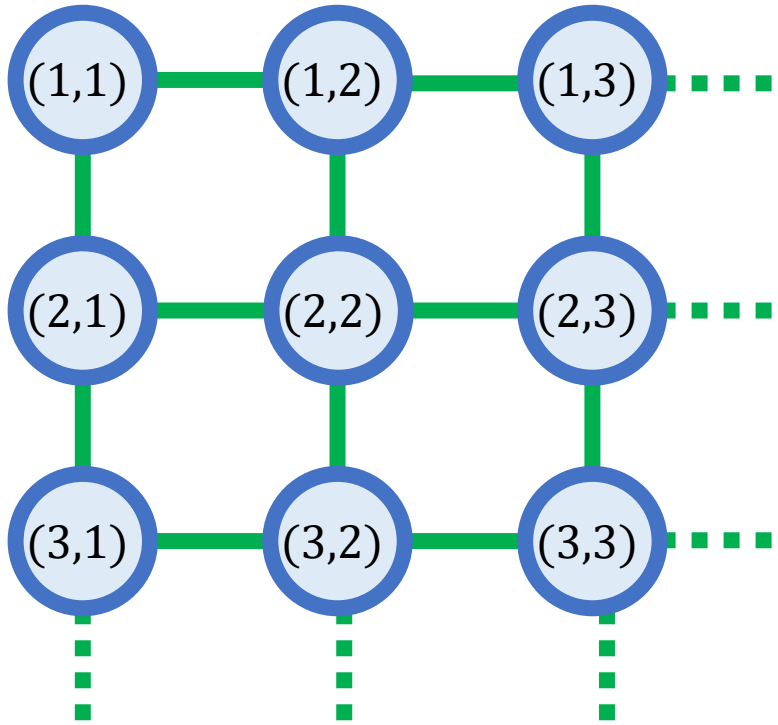
In the next lectures

- Some more advanced graph embeddings techniques

Learning from CNNs: images are graphs?

- **Previous lecture:** recognizing graph structures.
- Consider an image (greyscale, for simplicity).
  - It is a **graph**!
  - Nodes are pixels,
  - Nodes features are pixel values,
  - Edges connect neighboring pixels,
  - Edges are arranged in a regular grid.





Learning from CNNs: the need for convolutional operations

Lessons from the CNNs lectures:

- Keeping the structure of the image is essential: flattening an image and using Dense/FC layers does not work well.
- **Reason:** pixel values taken independently have little to no meaning.
- **Lesson:** the structure of the image matters!

→ **CNN solution: use convolutional operations, which help identify structural elements in the image.**

Learning from CNNs: the need for convolutional operations

- **For graphs, same idea:** nodes on their own have little meaning, the graph structure matters.
- **Observation (Homophily property of graphs):** Connected nodes in a graph tend to have similar features and labels.

Learning from CNNs: the need for convolutional operations

- **For graphs, same idea:** nodes on their own have little meaning, the graph structure matters.
- **Observation (Homophily property of graphs):** Connected nodes in a graph tend to have similar features and labels.
- This observation holds for images!
  - Connected pixels often have close values/colors.
  - Convolution allows to identify parts of the image where the homophily property does not hold.
  - (drastic shifts in pixel color/values = edges and corners)

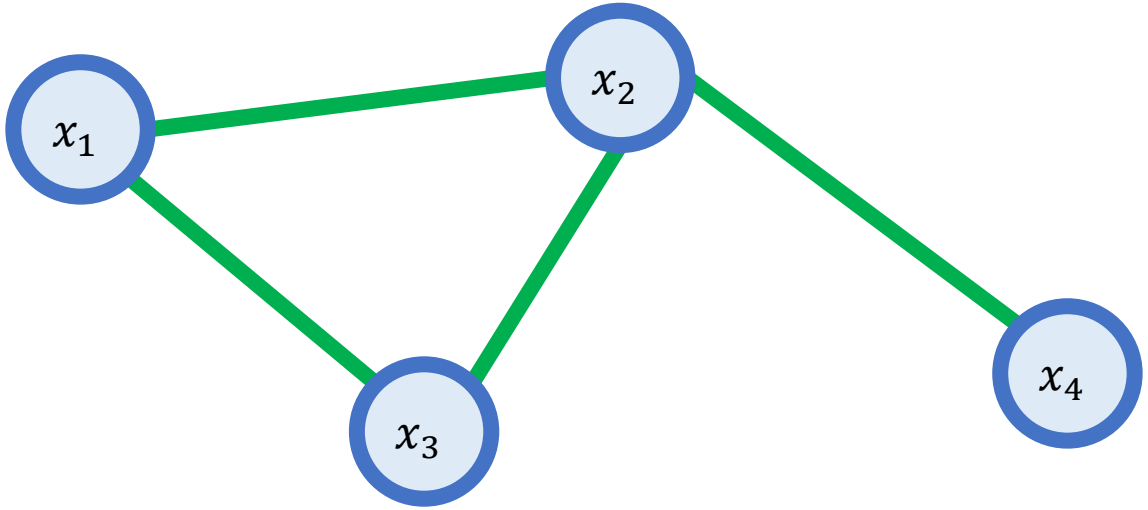
Learning from CNNs: the need for convolutional operations

- **For graphs, same idea:** nodes on their own have little meaning, the graph structure matters.
- **Observation (Homophily property of graphs):** Connected nodes in a graph tend to have similar features and labels.
- This observation holds for images!
  - Connected pixels often have close values/colors.
  - Convolution allows to identify parts of the image where the homophily property does not hold.
  - (drastic shifts in pixel color/values = edges and corners)

→ Key question: what would be the equivalent of an image convolution applied to a (not necessarily regular) graph?

Our toy example

- For demonstration purposes, let us consider the following graph.

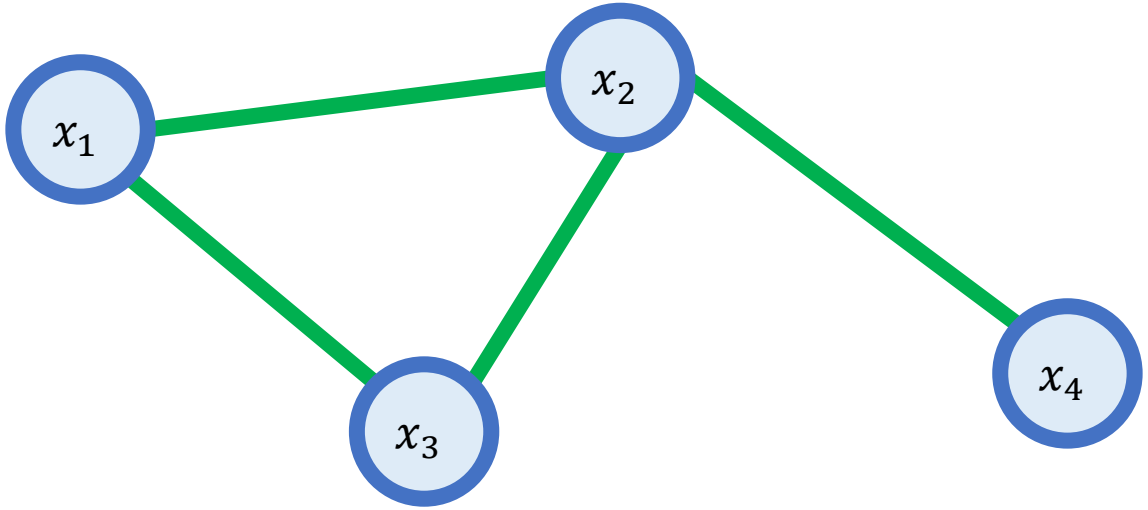


- Its adjacency matrix can then be defined as A.

```
# Adjacency matrix
A = np.matrix([
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 0],
    [0, 1, 0, 0]],
    dtype=float)
print("A:\n", A)
```

```
A:
 [[0. 1. 1. 0.]
 [1. 0. 1. 1.]
 [1. 1. 0. 0.]
 [0. 1. 0. 0.]]
```

Our toy example



- For demonstration purposes, let us consider the following graph.

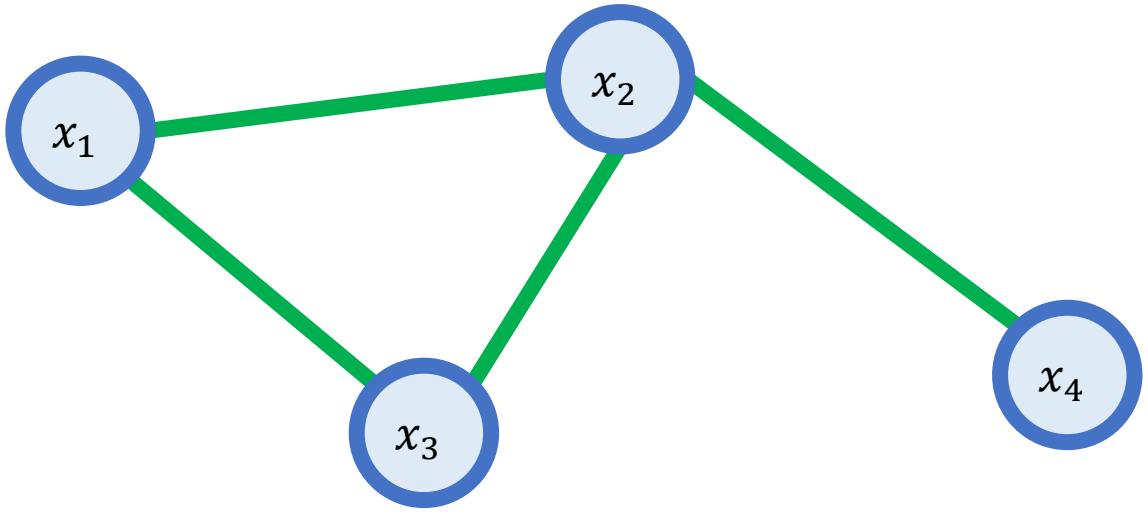
- Its adjacency matrix can then be defined as A.
- For demonstration, let us assume that each node x_i has a feature vector h_i of two elements:

$$h_i = (i, -i)$$

```
# Feature matrix
H = np.matrix([[i+1, -i-1] for i in range(A.shape[0])])
print("H:\n", H)
```

```
H:
 [[ 1 -1]
 [ 2 -2]
 [ 3 -3]
 [ 4 -4]]
```

Graph Convolution #1



- **Definition (graph convolution – nearby features only):**
  The graph convolution operation is defined as the multiplication

$$\boldsymbol{H' = AH}$$

With A the adjacency matrix of the graph and H the nodes features matrix of the graph.

- For each node x_i, it consists of the sum of its adjacent nodes features $\left(h_j\right)_{j \in neighbourhood(x_i)}$.

```
# Feature matrix
H = np.matrix([[i+1, -i-1] for i in range(A.shape[0])])
print("H:\n", H)
```

```
H:
 [[ 1 -1]
 [ 2 -2]
 [ 3 -3]
 [ 4 -4]]
```

```
# Identity function as activation
def f_identity(x):
    return x
```

```
# Propagation rule
H_next = f_identity(A*H)
print("H_next:\n", H_next)
```

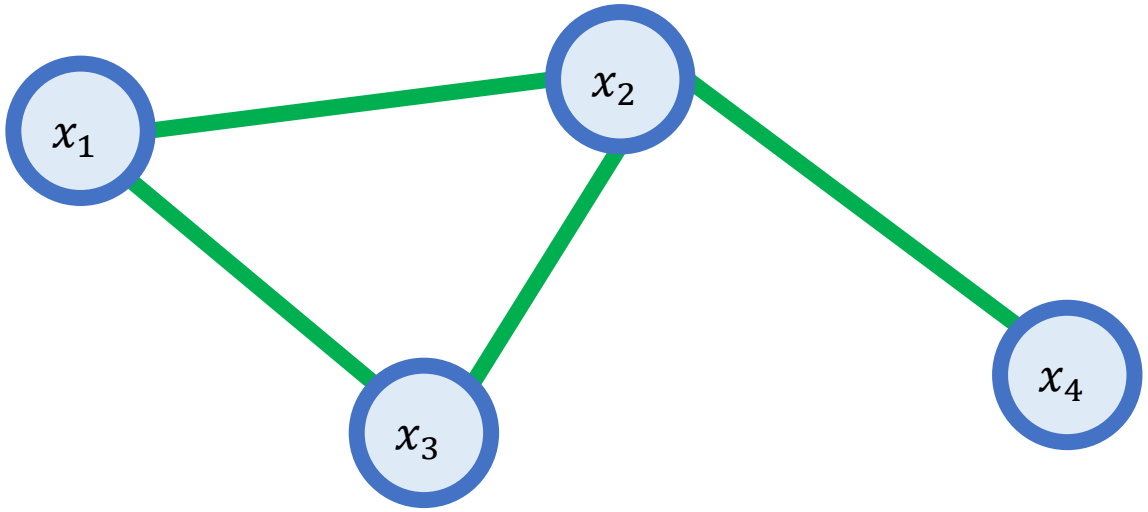
```
H_next:
 [[ 5. -5.]
 [ 8. -8.]
 [ 3. -3.]
 [ 2. -2.]]
```

Graph Convolution #2

- **Definition (graph convolution – self and nearby features):**
  The graph convolution operation is defined as the multiplication

$$\boldsymbol{H'_2} = \hat{\mathrm{A}}\boldsymbol{H}\text{, with } \hat{\mathrm{A}} = \boldsymbol{A} + \boldsymbol{I}$$

With A the adjacency matrix of the graph, H the nodes features matrix of the graph.




```
# Identity matrix
I = np.matrix(np.eye(A.shape[0]))
# Define A_hat as A + I
A_hat = A + I
print("A:\n", A)
print("\n")
print("A_hat:\n", A_hat)
```

```
A:
 [[0. 1. 1. 0.]
 [1. 0. 1. 1.]
 [1. 1. 0. 0.]
 [0. 1. 0. 0.]]


A_hat:
 [[1. 1. 1. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 0.]
 [0. 1. 0. 1.]]
```

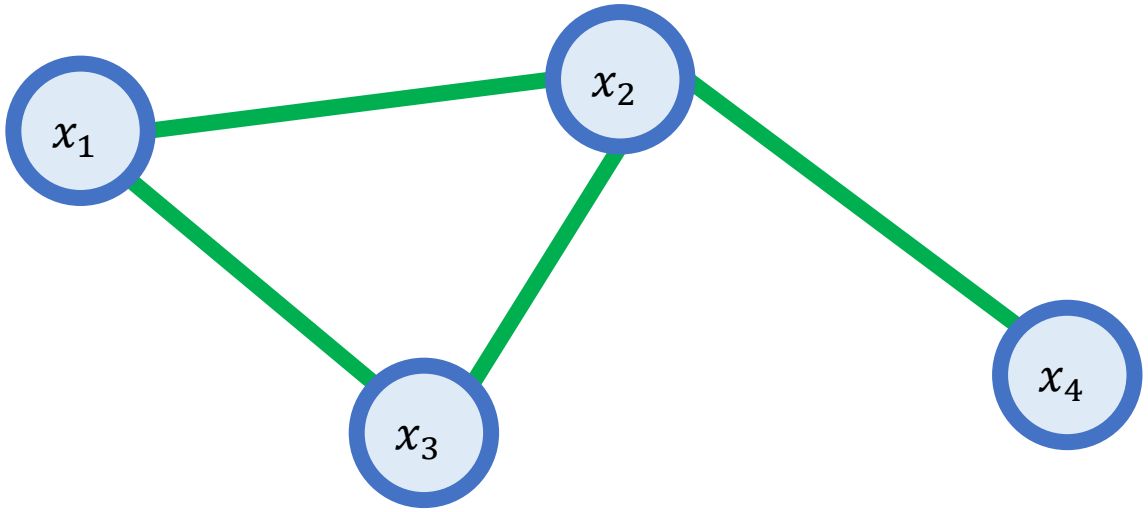
Graph Convolution #2

- **Definition (graph convolution – self and nearby features):** The graph convolution operation is defined as the multiplication

$$\boldsymbol{H'_2} = \hat{\mathrm{A}}\boldsymbol{H}\text{, with } \hat{\mathrm{A}} = \boldsymbol{A} + \boldsymbol{I}$$

With A the adjacency matrix of the graph, H the nodes features matrix of the graph.

- For each node x_i, it consists of the sum of its own features h_i and adjacent nodes features $(h_j)_{j \in neighbourhood(x_i)}$.




```
# Propagation rule
H_next2 = f_identity(A_hat*H)
print("H_next:\n", H_next)
print("\n")
print("H_next2:\n", H_next2)
```

```
H_next:
 [[ 5. -5.]
 [ 8. -8.]
 [ 3. -3.]
 [ 2. -2.]]


H_next2:
 [[  6.  -6.]
 [ 10. -10.]
 [  6.  -6.]
 [  6.  -6.]]
```

Image Convolution vs. Graph Convolution

- **Reminder (image convolution – 3x3 kernel, all kernel elements are ones, 1 channel):**

After the convolution operation, the result for pixel (i, j) is:

$$H_{i,j} = \frac{1}{9} \sum_{k_1=i-1}^{k_1=i+1} \sum_{k_2=j-1}^{k_2=j+1} x_{k_1,k_2}$$

With $x_{i,j}$ the features of pixel (i, j).

Weighted sum of self and nearby pixel values

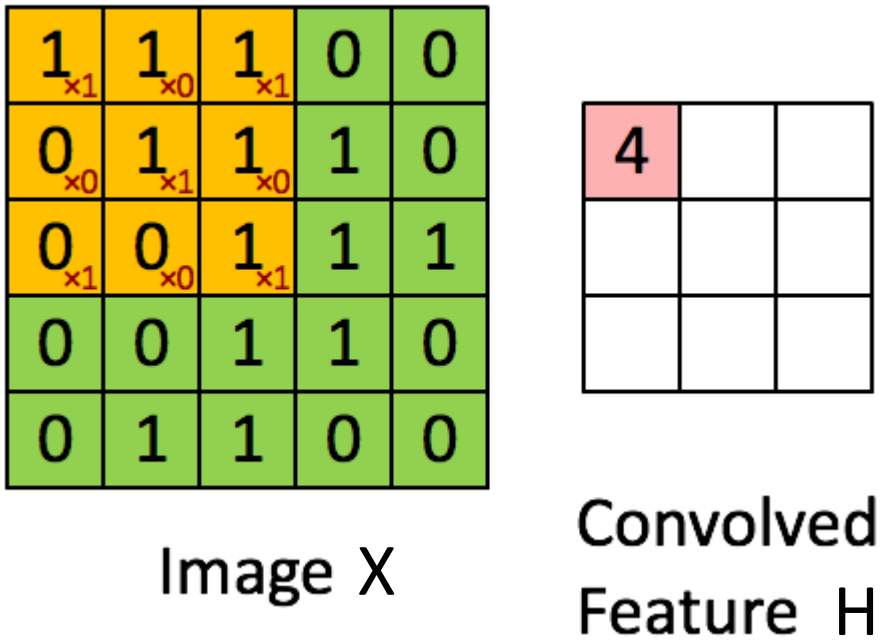


Image Convolution vs. Graph Convolution

- **Definition (graph convolution – self and nearby features):**
  The **graph convolution operation** is defined as the multiplication

$$\boldsymbol{H'} = \hat{\mathrm{A}}\boldsymbol{H}\textbf{, with } \hat{\mathrm{A}} = \boldsymbol{A} + \boldsymbol{I}$$

The operation gives, for node x_i:

$$H'_i = \sum_{j \in neighbours(x_i)} h_j$$

$$H'_i = \sum_{j\ s.t.\ (i,j) \in E} h_j$$

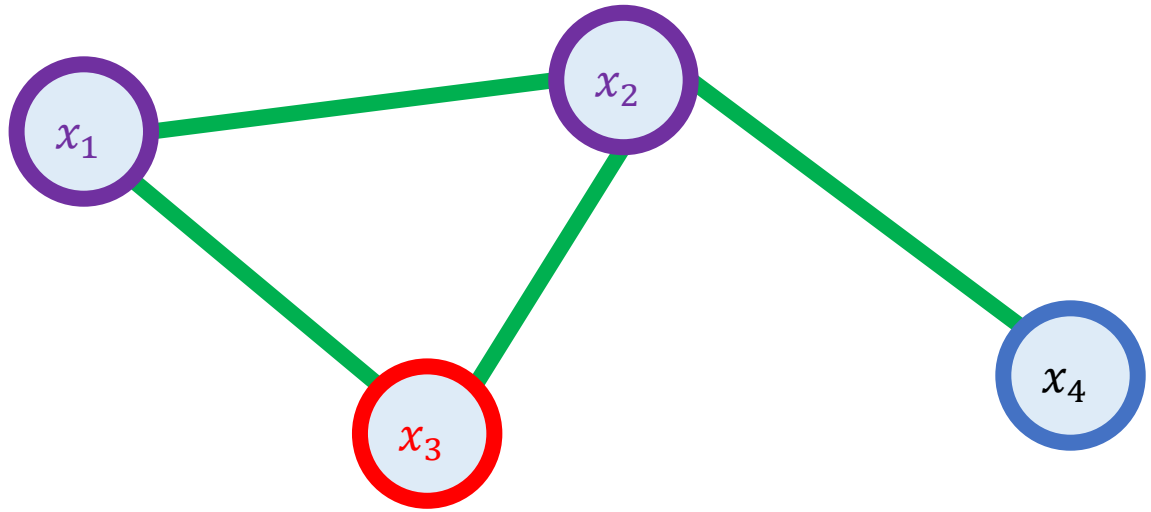
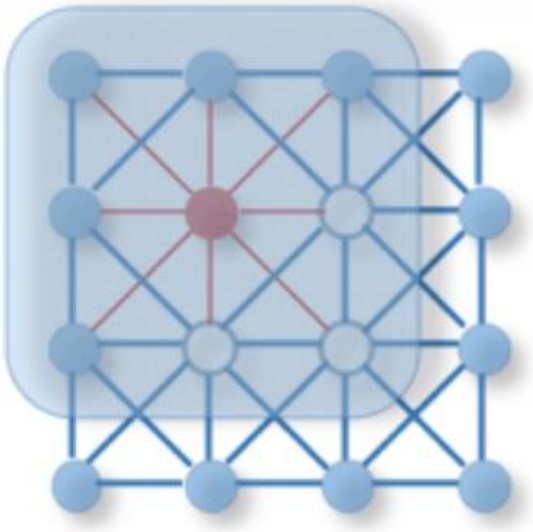
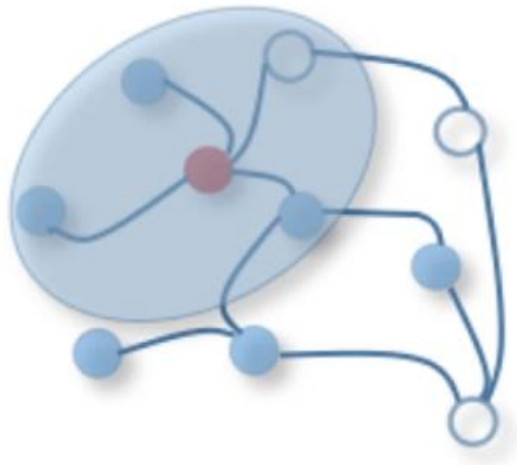


Image Convolution vs. Graph Convolution



(a) 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes a weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size.



(b) Graph Convolution. To get a hidden representation of the red node, one simple solution of graph convolution operation takes the average value of node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size.

Image Convolution vs. Graph Convolution

- **Definition (graph convolution – self and nearby features):**
  The **graph convolution operation** is defined as the multiplication

$$\boldsymbol{H'} = \hat{\mathrm{A}}\boldsymbol{H}\text{, with } \hat{\mathrm{A}} = \boldsymbol{A} + \boldsymbol{I}$$

The operation gives, for node x_i:

$$H'_i = \sum_{j \in neighbours(x_i)} h_j$$

$$H'_i = \sum_{j \; s.t. \; (i,j) \in E} h_j$$

- **Definition (image convolution – 3x3 kernel, 1 channel):**

After the convolution operation, the result for pixel (i, j) is:

$$H = \frac{1}{9} \sum_{k_1=i-1}^{k_1=i+1} \sum_{k_2=j-1}^{k_2=j+1} x_{k_1,k_2}$$

With $x_{i,j}$ the features of pixel (i, j).

Weighted sum of self and nearby pixel values

Image Convolution vs. Graph Convolution

- **Definition (graph convolution – self and nearby features):**
  The **graph convolution operation** is defined as the multiplication

$$\boldsymbol{H'} = \hat{\mathrm{A}}\boldsymbol{H}\text{, with } \hat{\mathrm{A}} = \boldsymbol{A} + \boldsymbol{I}$$

The operation gives, for node x_i:

$$H'_i = ? \sum_{j\ s.t.\ (i,j)\in E} h_j$$

→ **Very similar, just missing a normalizing factor!**

- **Definition (image convolution – 3x3 kernel, 1 channel):**

After the convolution operation, the result for pixel (i, j) is:

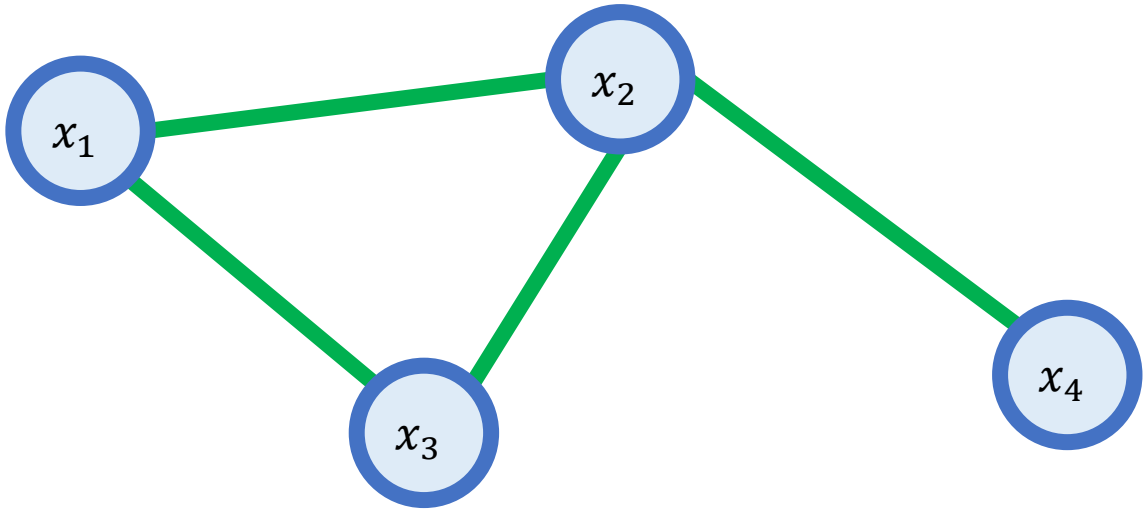
$$H = \frac{1}{9} \sum_{k_1=i-1}^{k_1=i+1} \sum_{k_2=j-1}^{k_2=j+1} x_{k_1,k_2}$$

With $x_{i,j}$ the features of pixel (i, j).

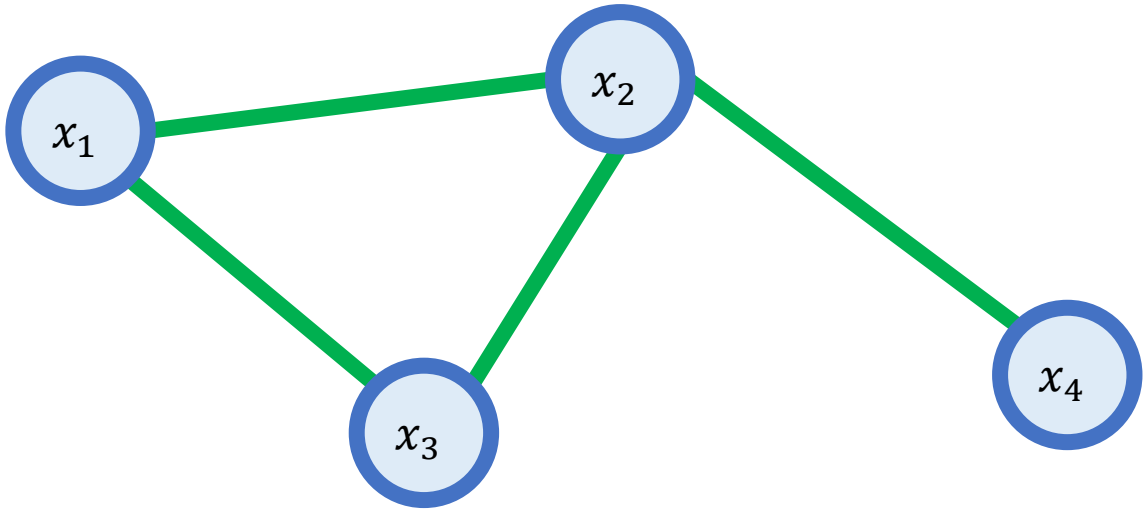
Weighted sum of self and nearby pixel values

Normalizing

- **Question:** What is the name of the normalizing factor we are looking for?



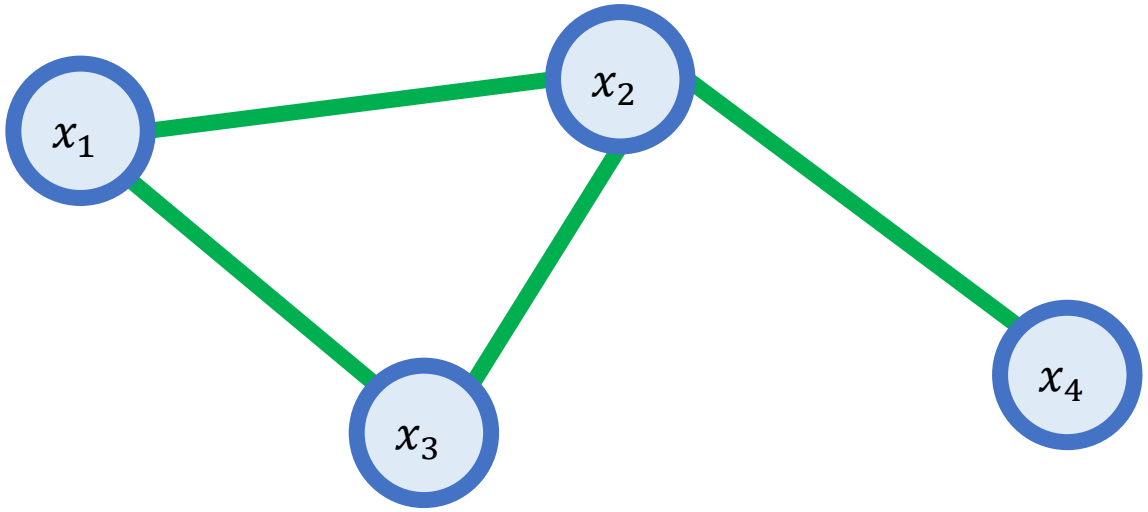
Normalizing (degrees)



- **Question:** What is the name of the normalizing factor we are looking for?
- Normalizing = dividing by the degree of node x_i, plus one, i.e.

$$\boldsymbol{d_{ii} + 1}$$

Normalizing (degrees)



- **Definition:** We denote as $\widehat{\boldsymbol{D}}$, the diagonal matrix of degrees plus one.
- It can be computed from $\hat{\mathrm{A}}$, by summing its rows and reporting the values on the diagonal.
- Or, even simpler, $\widehat{\boldsymbol{D}} = \boldsymbol{D} + \boldsymbol{I}$.

In [14]:
```
1 # Define D_hat as the degree matrix of A_hat
2 D_hat = np.array(np.sum(A_hat, axis=0))[0]
3 D_hat = np.matrix(np.diag(D_hat))
4 print("D_hat:\n", D_hat)
```

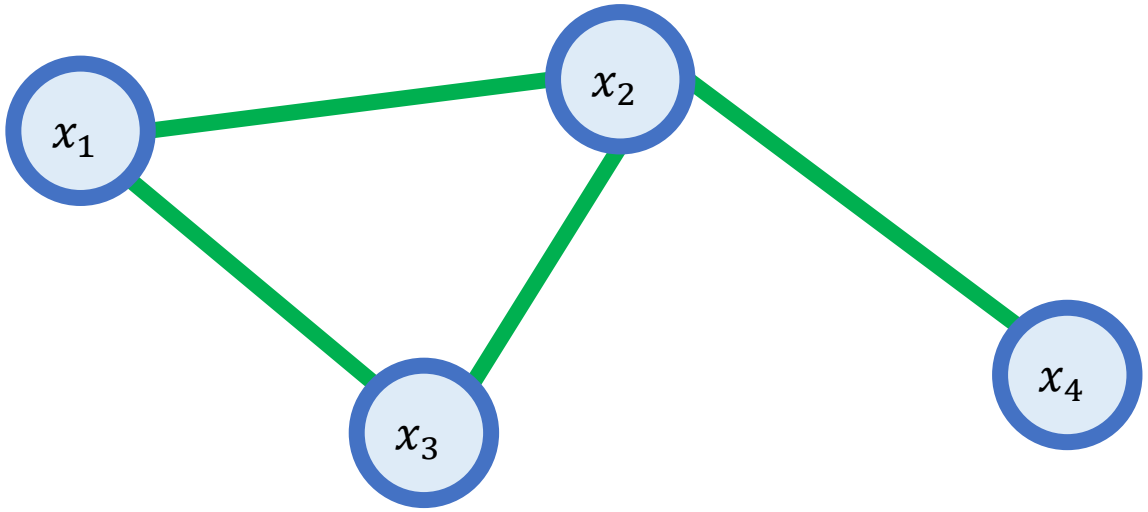
```
D_hat:
 [[3. 0. 0. 0.]
 [0. 4. 0. 0.]
 [0. 0. 3. 0.]
 [0. 0. 0. 2.]]
```

Normalizing (degrees)

- **Definition:** We denote as $\widehat{\boldsymbol{D}}$, the diagonal matrix of degrees plus one.

- **Definition (normalizing graph convolution matrix):** The normalizing graph convolution matrix $\widehat{\boldsymbol{N}}$ is defined as

$$\widehat{\boldsymbol{N}} = \widehat{\boldsymbol{D}}^{-1}\hat{\mathrm{A}}$$

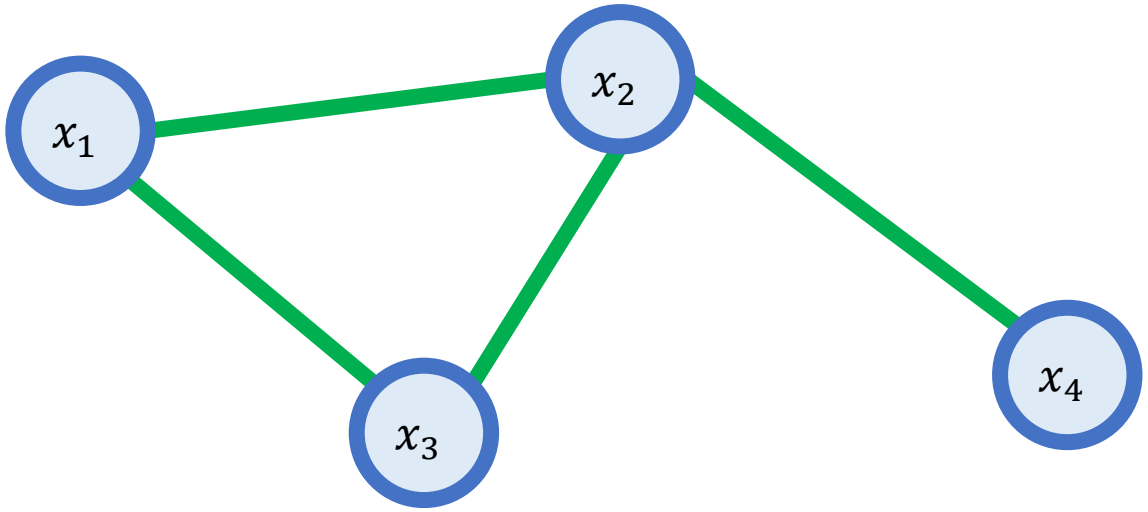



```
A_hat:
 [[1. 1. 1. 0.]
 [1. 1. 1. 1.]
 [1. 1. 1. 0.]
 [0. 1. 0. 1.]]
```

```
In [15]:
1 # Define N_hat (normalization)
2 N_hat = (D_hat**-1)*A_hat
3 print("N_hat:\n", N_hat)
```

```
N_hat:
 [[0.33333333 0.33333333 0.33333333 0.        ]
 [0.25       0.25       0.25       0.25      ]
 [0.33333333 0.33333333 0.33333333 0.        ]
 [0.         0.5        0.         0.5       ]]
```

Normalizing (degrees)



- **Definition (normalized graph convolution):** The **normalized graph convolution** operation is defined as follows:

$$\boldsymbol{H' = \hat{N}H}$$

- Very similar to previous convolutions, but less variance in the values.

```
# Update propagation rule
H_next3 = f_identity(N*H)
print("H_next3:\n", H_next3)
print("H_next2:\n", H_next2)
```

```
H_next3:
 [[ 2.5         -2.5       ]
 [ 2.66666667 -2.66666667]
 [ 1.5         -1.5       ]
 [ 2.          -2.        ]]
H_next2:
 [[  6.  -6.]
 [ 10. -10.]
 [  6.  -6.]
 [  6.  -6.]]
```

Adding weights to graph convolutions

- **Image convolutions:** could define weights, i.e. values for the kernel matrix, to identify key features (horizontal/vertical edges, corners, etc.) in the image. Later, we trained these weights.
- **Graph convolutions, with weights:** similarly, we could define weights $\boldsymbol{W}$ to assign to our graph convolution operation.

Adding weights to graph convolutions

- **Image convolutions:** could define weights, i.e. values for the kernel matrix, to identify key features (horizontal/vertical edges, corners, etc.) in the image. Later, we trained these weights.
- **Graph convolutions, with weights:** similarly, we could define weights $\boldsymbol{W}$ to assign to our graph convolution operation.
- **Propagation rule:** while we are at it, the relationship

$$\boldsymbol{H} = \boldsymbol{f}(\boldsymbol{A}, \boldsymbol{X}, \boldsymbol{W})$$

can be used to define a propagation rule. And we can later on, train our weights $\boldsymbol{W}$ as well.

→ **Next logical step: define a graph convolutional layer.**

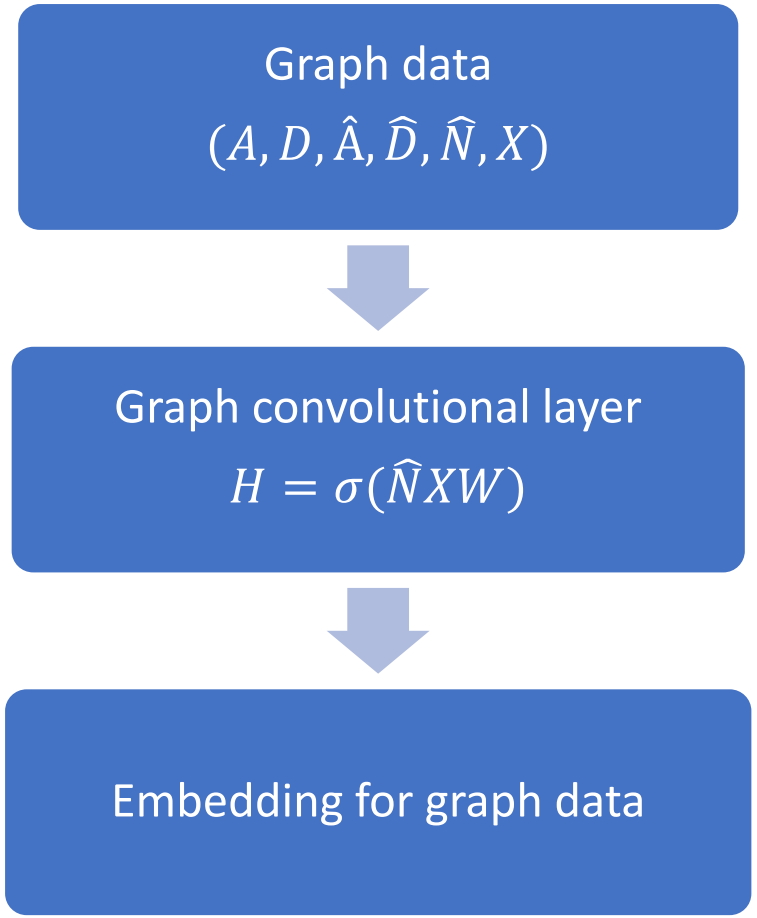
From graph convolution to graph convolutional layers

- **Definition (graph convolutional layer):** We can define a graph convolutional layer by using the following propagation rule.

$$\boldsymbol{H = \sigma(\hat{N}XW)}$$

With $\boldsymbol{\sigma}$ an **activation function** (so far we used identity), and $\boldsymbol{W}$ a **weight matrix**, which we will train later on.

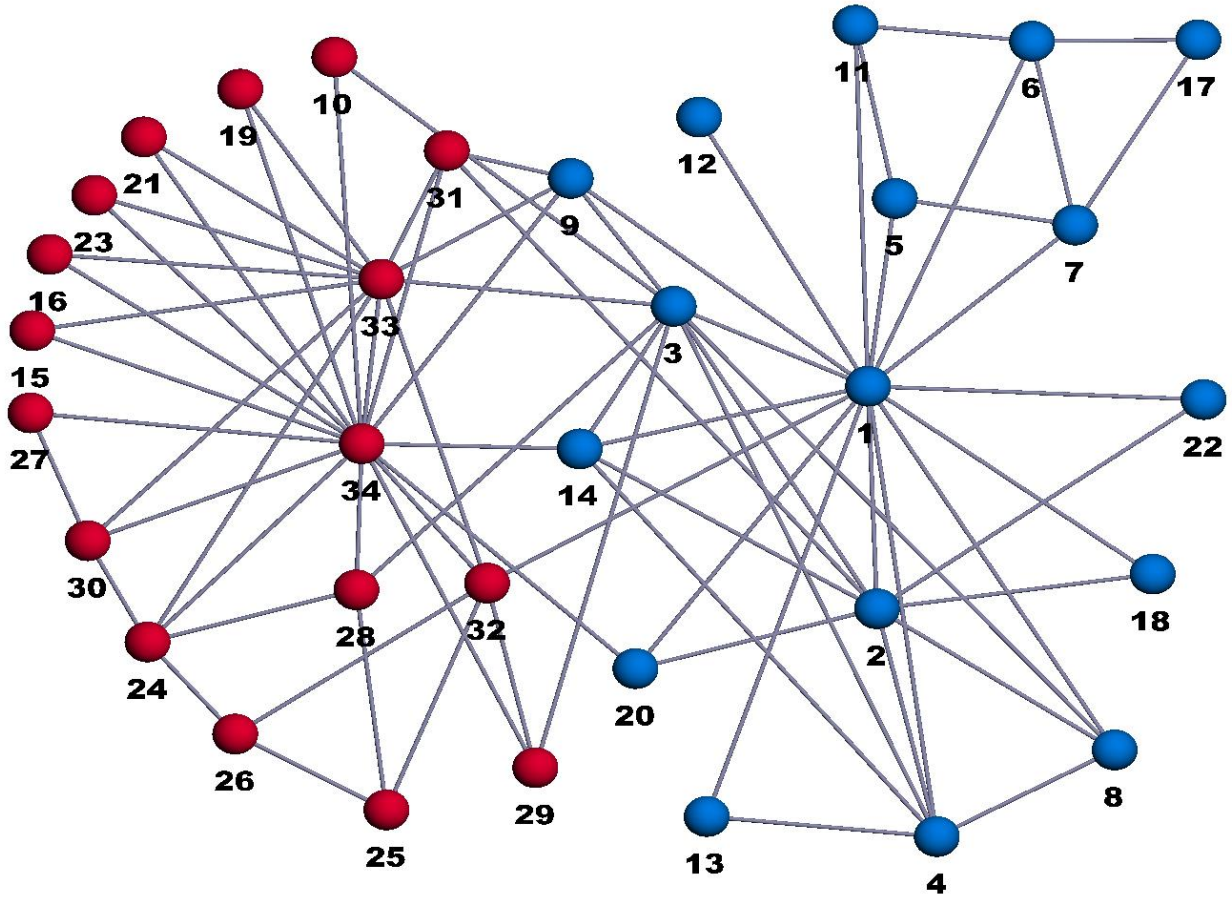
- This allows to define embeddings for our graph data!



A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- 34 members, as nodes, indexed from 1 to 34.
- Mr. Hi, the owner is node 1.
- The Officer/Trainer is node 34.
- All other nodes are members.
- Edges define friendships between individuals.
- **Question:** If Mr. Hi and the officer/trainer decide to fight, what will be the teams?



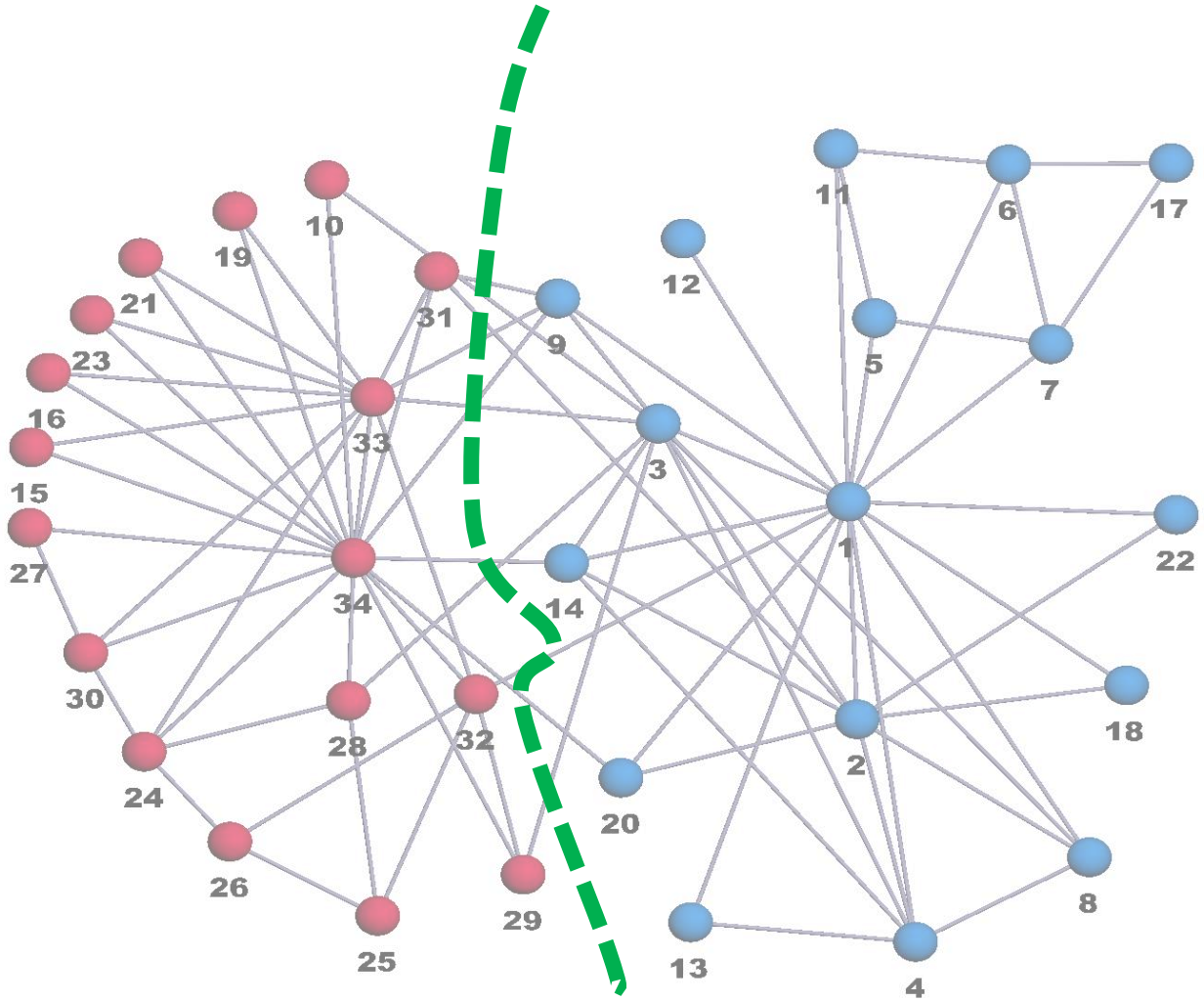
[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- It consists of a binary classification problem.
- Separate the nodes (as with any classification problem).
- Need an embedding to represent the graph data, as a matrix/vector of some sort.

→ **Graph convolutional layer**



[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

In [25]:
```
# Import Zachary's karate club dataset
from networkx import karate_club_graph, to_numpy_matrix
zkc = karate_club_graph()
```

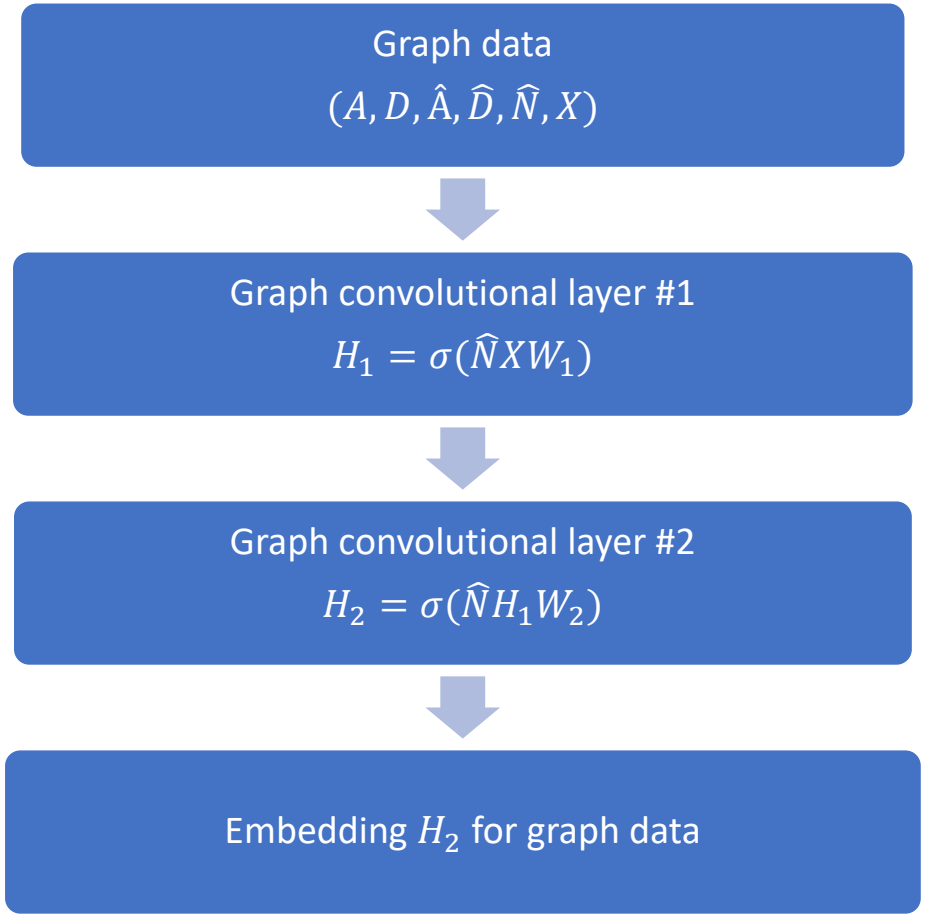
In [27]:
```
nodes_index_list = []
for node_key, node_value in zkc._node.items():
    print("Node index: {}, node labels: {}".format(node_key, node_value))
    nodes_index_list.append(node_key)
```

```
Node index: 0, node labels: {'club': 'Mr. Hi'}
Node index: 1, node labels: {'club': 'Mr. Hi'}
Node index: 2, node labels: {'club': 'Mr. Hi'}
Node index: 3, node labels: {'club': 'Mr. Hi'}
Node index: 4, node labels: {'club': 'Mr. Hi'}
Node index: 5, node labels: {'club': 'Mr. Hi'}
Node index: 6, node labels: {'club': 'Mr. Hi'}
Node index: 7, node labels: {'club': 'Mr. Hi'}
Node index: 8, node labels: {'club': 'Mr. Hi'}
Node index: 9, node labels: {'club': 'Officer'}
Node index: 10, node labels: {'club': 'Mr. Hi'}
```

Spectral graph embedding

Our setup (Spectral graph embedding, with graph convolutional layers):

- Two graph convolutional layers
- ReLU as activation function
- Normalized graph convolution



Spectral graph embedding

Our setup (Spectral graph embedding, with graph convolutional layers):

- Two graph convolutional layers
- ReLU as activation function
- Normalized graph convolution
- $W_1.shape = [34, 4]$
- $W_2.shape = [4, 2]$
- Initialized as uniformly random

In [35]:

```
# Initialize weights for two layers, using Normal(0,1) random variables
W1 = np.random.normal(loc = 0, scale = 1, size = (number_of_nodes, 4))
W2 = np.random.normal(loc = 0, size = (W1.shape[1], 2))
print("W1:\n", W1)
print("\n")
print("W2:\n", W2)
```

Spectral graph embedding

Our setup (Spectral graph embedding, with graph convolutional layers):

- Two graph convolutional layers
- ReLU as activation function
- Normalized graph convolution
- $W_1.shape = [34, 4]$
- $W_2.shape = [4, 2]$
- Initialized as uniformly random

In [35]:
```
# Initialize weights for two layers, using Normal(0,1) random variables
W1 = np.random.normal(loc = 0, scale = 1, size = (number_of_nodes, 4))
W2 = np.random.normal(loc = 0, size = (W1.shape[1], 2))
print("W1:\n", W1)
print("\n")
print("W2:\n", W2)
```

In [36]:
```
# Propagation rule as a function (normalized ReLU)
def gcn_normalized_layer(A_hat, D_hat, X, W):
    return f_relu((D_hat**-1)*A_hat*X*W)
```

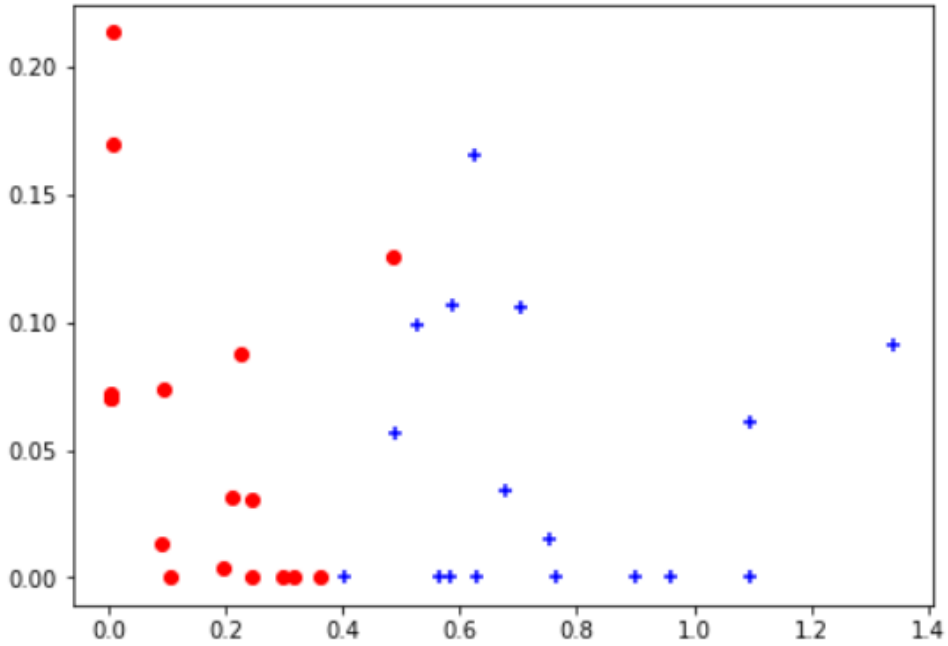
In [37]:
```
# Compute output for GCN normalized layer 1
H_layer1 = gcn_normalized_layer(A_hat, D_hat, X, W1)
# Compute output for GCN normalized layer 2
H_layer2 = gcn_normalized_layer(A_hat, D_hat, H_layer1, W2)
print("H_layer1:\n", H_layer1)
print("H_layer2:\n", H_layer2)
```

Spectral graph embedding

Spectral graph embedding

- Sometimes it works…

→ (Ready for classification!)

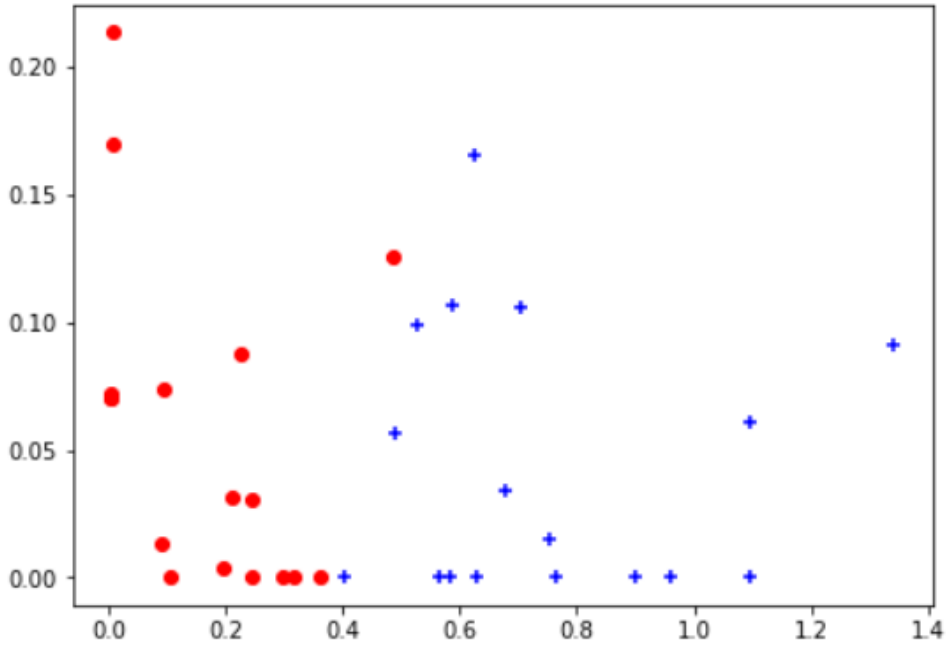


Spectral graph embedding

Spectral graph embedding

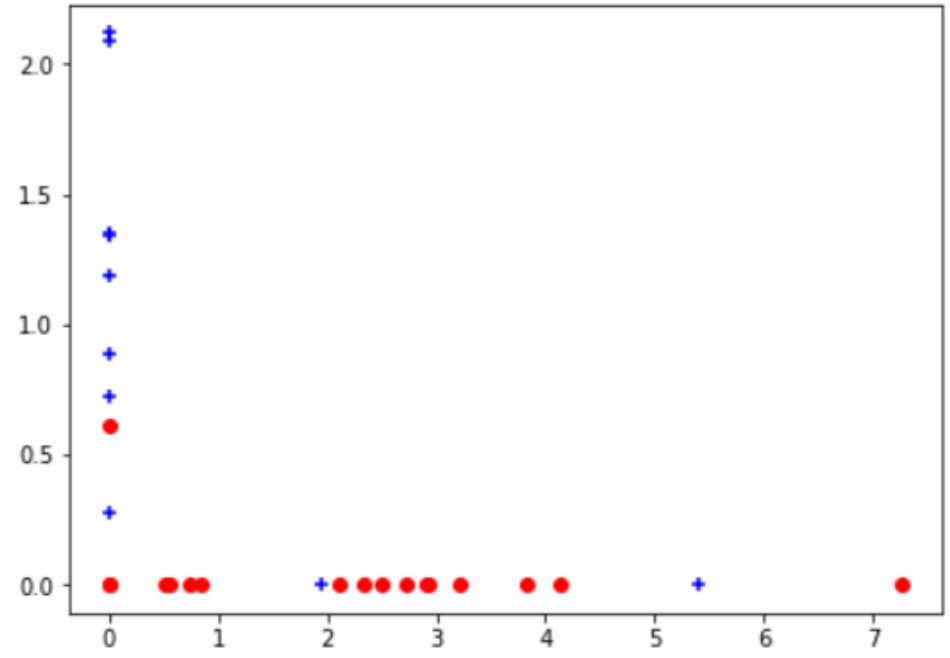
- Sometimes it works…

→ (Ready for classification!)



- Sometimes it does not.

→ (Oh crap…)



- That's why you need to **train** it (and adjust the weights in W).

Graph convolutional layer (Kipf)

- **Definition (graph convolutional layer, with conventional spectral propagation rule):**

$$\boldsymbol{H' = \sigma\left(\hat{N}HW\right) = \sigma\left(\hat{D}^{-1}\hat{A}HW\right)}$$

With $\boldsymbol{\sigma}$ an **activation function** (so far we used identity), and $\boldsymbol{W}$ a **weight matrix**, which we will train later on.

Graph convolutional layer (Kipf)

- **Definition (graph convolutional layer, with conventional spectral propagation rule):**

$$\boldsymbol{H' = \sigma(\hat{N}HW) = \sigma(\hat{D}^{-1}\hat{A}HW)}$$

With $\boldsymbol{\sigma}$ an **activation function** (so far we used identity), and $\boldsymbol{W}$ a **weight matrix**, which we will train later on.

- **Definition (graph convolutional layer, with Kipf spectral propagation rule as in the paper *[Kipf]*):**

$$\boldsymbol{H' = \sigma(\hat{N}^{kipf}HW)} = \boldsymbol{\sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}HW)}$$

- Slightly better than $\hat{D}^{-1}\hat{A}$, especially in oriented graphs.
- $\boldsymbol{M = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}}$ is called the **symmetric Laplacian of the graph**.

[Kipf] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks.

Restricted

Graph convolutional layer (Kipf)

- **Definition (graph convolutional layer, with conventional spectral propagation rule):**

$$\boldsymbol{H = \sigma(\hat{N}XW) = \sigma(\hat{D}^{-1}\hat{A}XW)}$$

With $\boldsymbol{\sigma}$ an **activation function** (so far we used identity), and $\boldsymbol{W}$ a **weight matrix**, which we will train later on.

→ Question: why is matrix $\boldsymbol{M}$ more interesting mathematically speaking?

- **Definition (graph convolutional layer, with Kipf spectral propagation rule as in the paper *[Kipf]*):**

$$\boldsymbol{H = \sigma(\hat{N}^{kipf}XW) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}XW)}$$

- Slightly better than $\hat{D}^{-1}\hat{A}$, especially in oriented graphs.
- $\boldsymbol{M = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}}$ is called the symmetric Laplacian of the graph.

Graph convolutional layer (Kipf) vs. “Normal” graph convolutional layer

- Our “normal” graph convolutional layer

$$\hat{D}^{-1}\hat{A}H = \sum_{j\ s.t.\ (i,j)\in E} \frac{h_j}{d_i + 1} = \sum_{j=1}^{N} \frac{h_j A_{ij}}{d_i + 1}$$

- Kipf layer is slightly different, as it no longer amounts to mere averaging of neighboring nodes

$$\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H = \sum_{j=1}^{N} \frac{h_j A_{ij}}{\sqrt{d_i + 1}\sqrt{d_j + 1}}$$

Takes into account the degree of neighbors, as extra structural information: helps low-degree neighbors to provide more useful information than high-degree neighbors.

A quick word on Laplacians (out of class)

Definitions (some Laplacians): Given a graph G, with adjacency matrix A and degree matrix D, we can define the following matrices:

- The Laplacian matrix:

$$L = D - A$$

- The random walk Laplacian

$$L_{rw} = I - D^{-1}A = D^{-1}L$$

- The symmetric Laplacian

$$L_{sym} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$$

These matrices have interesting math properties in graph theory, but we will leave those to the curious reader.

Why did we use Laplacians for our spectral embedding layers? (intuition, out of class)

- **Reason:** The spectral properties (eigenvalues and eigenvectors) of the Laplacian matrices give interesting insights on the structure/connectivity of the graph.

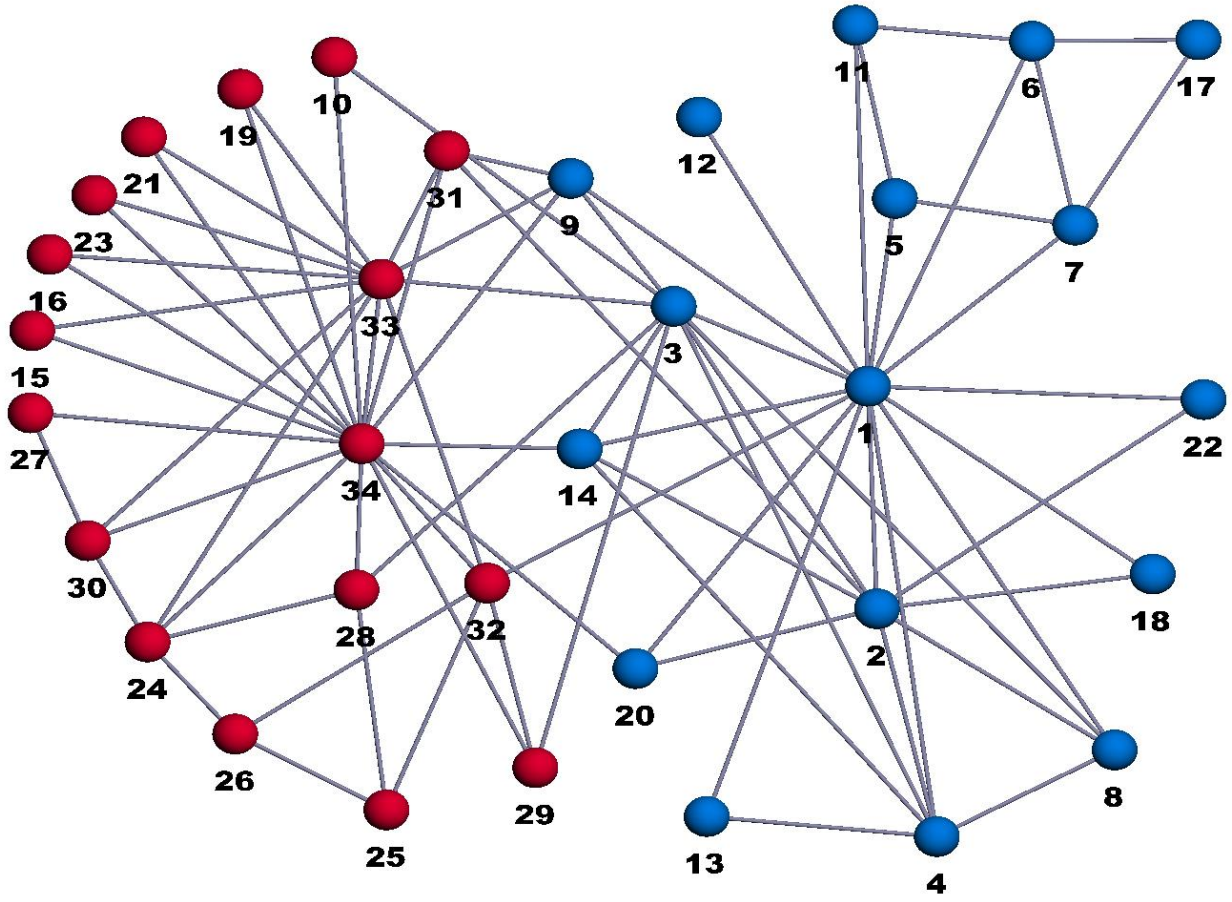
- See challenges 1-2-3 on the “normal” Laplacian matrix properties in previous lecture for some examples!

- The variations of the Laplacians give different properties (see Lecture Notes with challenge answers given on W10S1 if curious!)

A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- 34 members, as nodes, indexed from 1 to 34.
- Mr. Hi, the owner is node 1.
- The Officer/Trainer is node 34.
- All other nodes are members.
- Edges define friendships between individuals.
- **Question:** If Mr. Hi and the officer/trainer decide to fight, what will be the teams?



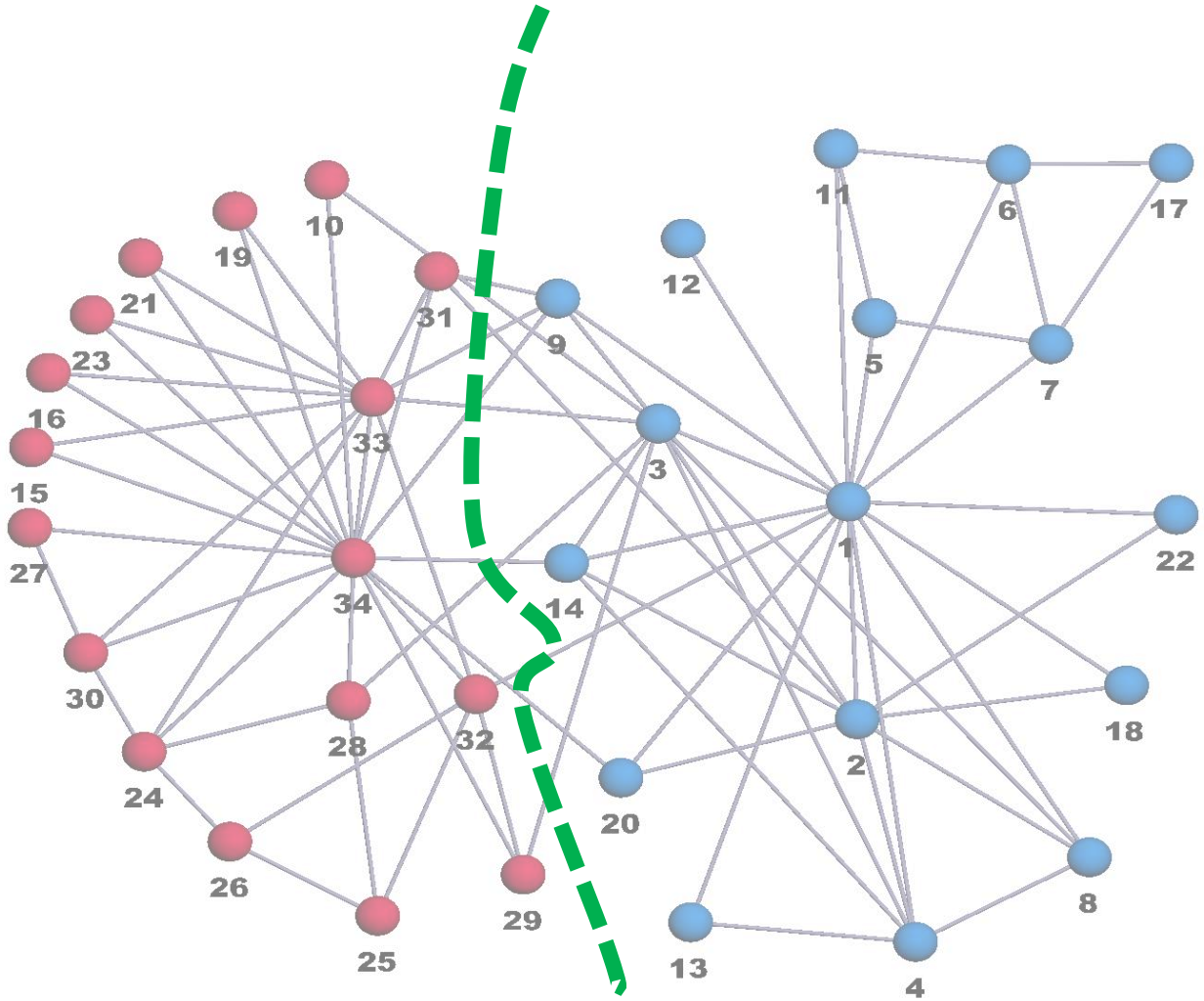
[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

A more sophisticated toy example

Zachary Karate Fight club *[ZKC]*:

- It consists of a binary classification problem.
- Separate the nodes (as with any classification problem).
- Need an embedding to represent the graph data, as a matrix/vector of some sort.

→ **Graph convolutional layer**



[ZKC] Zachary, W. W. (1977). An information flow model for conflict and fission in small groups.

Graph dataset: import and information extraction (nodes, edges, matrices)

- We will import our dataset from the **networkx** library.

Then, we design two functions to extract:

- The nodes information (names, features, labels)
- The nodes are indexed (0-33), they have no features and have a single label (0 = Mr. Hi, 1 = Officer/Trainer)
- The edges are defined as tuples (i, j).
- The edges have no features and labels.
- We also reconstruct the adjacency matrix and degree matrix.

In [35]:

```python
def define_nodes_parameters(G):

    # Initialize nodes parameters
    nodes_parameters = {}

    # Nodes number
    nodes_parameters['nodes_number'] = 0

    # Nodes names
    nodes_parameters['nodes_names'] = []

    # Nodes features
    nodes_parameters['nodes_features'] = []

    # Labels list
    nodes_parameters['labels_list'] = []

    # Nodes labels
    nodes_parameters['nodes_labels'] = []

    # Read nodes from graph
    for node_key, node_value in G._node.items():
        nodes_parameters['nodes_number'] += 1
        nodes_parameters['nodes_names'].append(node_key)
        node_value = node_value['club']
        if not node_value in nodes_parameters['labels_list']:
            nodes_parameters['labels_list'].append(node_value)
        nodes_parameters['nodes_labels'].append(nodes_parameters['labels_list'].index(node_value))

    # Return
    return nodes_parameters
```

In [36]:

```python
nodes_parameters = define_nodes_parameters(G)
print(nodes_parameters)
```

```
{'nodes_number': 34, 'nodes_names': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30, 31, 32, 33], 'nodes_features': [], 'labels_list': ['Mr. Hi', 'Officer'], 'nodes_labels': [0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

In [5]:

```python
def define_edges_parameters(G, nodes_number):

    # Initialize edges parameters
    edges_parameters = {}

    # Edges number
    edges_parameters['edges_number'] = len(list(G.edges().keys()))

    # Edges names
    edges_parameters['edges_names'] = list(G.edges().keys())

    # Adjacency matrix
    edges_parameters['adjacency_matrix'] = np.zeros([nodes_number, nodes_number])

    # Degree matrix
    edges_parameters['degree_matrix'] = np.zeros([nodes_number, nodes_number])

    # Read edges and update adjacency matrix and degree matrix
    for edge in edges_parameters['edges_names']:
        node1, node2 = edge
        edges_parameters['adjacency_matrix'][node1, node2] = 1
        edges_parameters['adjacency_matrix'][node2, node1] = 1
        edges_parameters['degree_matrix'][node1, node1] += 1
        edges_parameters['degree_matrix'][node2, node2] += 1

    # Return
    return edges_parameters
```

In [6]:

```python
edges_parameters = define_edges_parameters(G, nodes_number = nodes_parameters['nodes_number'])
print(edges_parameters)
```

```
{'edges_number': 78, 'edges_names': [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 10), (0, 11), (0, 12),
(0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2), (1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 21), (1, 30), (2, 3), (2,
7), (2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (2, 32), (3, 7), (3, 12), (3, 13), (4, 6), (4, 10), (5, 6), (5, 10), (5, 16),
(6, 16), (8, 30), (8, 32), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32), (15, 33), (18, 32), (18, 33), (19, 33), (2
0, 32), (20, 33), (22, 32), (22, 33), (23, 25), (23, 27), (23, 29), (23, 32), (23, 33), (24, 25), (24, 27), (24, 31), (25, 31),
(26, 29), (26, 33), (27, 33), (28, 31), (28, 33), (29, 32), (29, 33), (30, 32), (30, 33), (31, 32), (31, 33), (32, 33)], 'adjac
ency_matrix': array([[0., 1., 1., ..., 1., 0., 0.],
       [1., 0., 1., ..., 0., 0., 0.],
       [1., 1., 0., ..., 0., 1., 0.],
       ...,
       [1., 0., 0., ..., 0., 1., 1.],
       [0., 0., 1., ..., 1., 0., 1.],
       [0., 0., 0., ..., 1., 1., 0.]]), 'degree_matrix': array([[16.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  9.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0., 10., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  6.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0., 12.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0., 17.]])}
```

```
# Adjacency matrix for Zachary graph
adj = np.array([[0,1,1,1,1,1,1,1,1,0,1,1,1,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,1,0,0],
                [1,0,1,1,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,1,0,0,0],
                [1,1,0,1,0,0,0,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0],
                [1,1,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,1,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,0,0],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,1],
                [0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1],
                [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1,1],
                [0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1],
                [1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,1,1],
                [0,0,1,0,0,0,0,0,1,0,0,0,0,0,1,1,0,0,1,0,1,0,1,1,0,0,0,0,0,1,1,1,0,1],
                [0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,0,0,1,1,1,0,1,1,0,0,1,1,1,1,1,1,1,0]])
A = torch.Tensor(adj)
```

Graph dataset: import and information extraction (nodes, edges, matrices)

- The only nodes with pre-established labels are the nodes 0 and 33.
- 0 = Mr. Hi, which always gets classified as class 0,
- 33 = Officer/Trainer, which always gets classified as class 1.
- Ground truth labels available for all nodes.
- -1 as current class means it is a member and needs to be classified

```
# Labels for all nodes
# (0 is admin, 34 is instructor, all others are -1 because they are currently unlabeled)
current = torch.tensor([0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, \
                        -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 1])
```

```
# Ground truth for reference
ground_truth = torch.tensor([0, 0, 0, 0 ,0 ,0 ,0, 0, 1, 1, 0 ,0, 0, 0, 1 ,1 ,0 ,0, \
                             1, 0, 1, 0 ,1 ,1, 1, 1, 1 ,1 ,1, 1, 1, 1, 1 ])
```

Building blocks for classifier

- Classification task on graph dataset (ZKC).
- **DL library:** Notebooks with implementations on both **PyTorch** and **MXNet** (very similar to PyTorch, Tensorflow, etc.)
- **Note:** we are showing this library so you can see that even though multiple libraries exist for DL, they more or less follow the same logic as PyTorch!

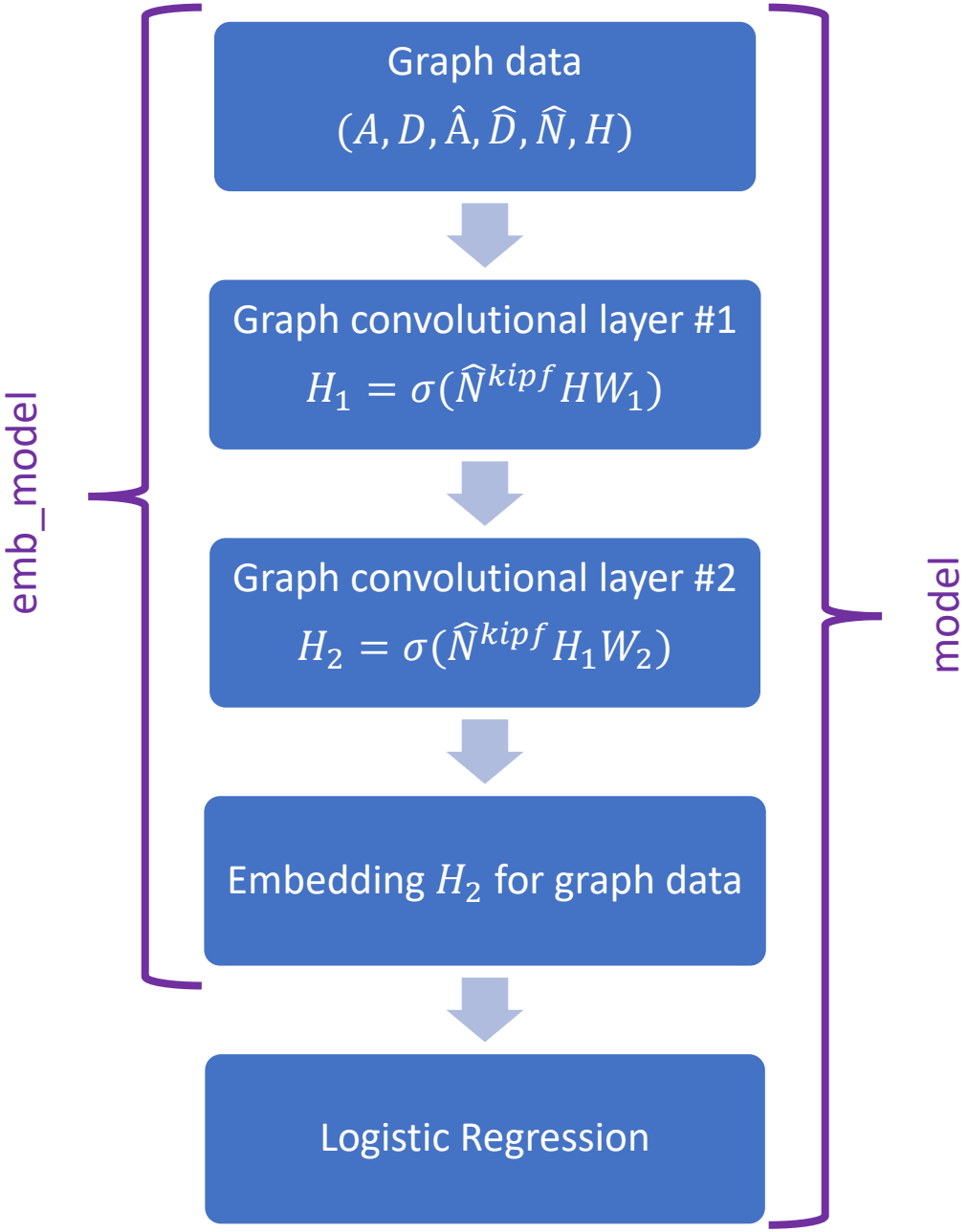
In [1]:

```
from collections import namedtuple

import matplotlib.pyplot as plt

import mxnet.ndarray as nd
from mxnet import autograd, init
from mxnet.initializer import One, Uniform, Xavier
from mxnet.gluon import HybridBlock, Trainer
from mxnet.gluon.loss import SigmoidBinaryCrossEntropyLoss
from mxnet.gluon.nn import HybridSequential, Activation
from mxnet.ndarray import array
from mxnet.ndarray import sum as ndsum
from mxnet.random import seed as mxn_seed

from networkx import read_edgelist, set_node_attributes
from networkx import karate_club_graph, to_numpy_matrix
from networkx import shortest_path_length

import numpy as np
from numpy import array
from numpy.random import seed as np_seed

from pandas import read_csv, Series

from sklearn.metrics import classification_report
```

Building blocks for classifier

- Classification task on graph dataset (ZKC).

- **Structure:**
  - 2 GCN layers (with Kipf propagation rule)
  - 1 Logistic Regression for classification



Building a basic GCN layer and model

- Create our own custom basic GCN layer block, using the PyTorch nn.Module class as superclass.
- Pass the adjacency matrix $\boldsymbol{A}$, compute $\hat{\mathrm{A}}$, $\widehat{\boldsymbol{D}}$, etc.
- Propagation rule: $\boldsymbol{H}' = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\mathbf{1}}\widehat{\boldsymbol{A}}\boldsymbol{H}\boldsymbol{W}\right)$

```python
class GCNConv_Layer(nn.Module):
    """
    Standard GCN convolution layer class
    """

    def __init__(self, adj, input_channels, output_channels):
        super().__init__()
        self.A_hat = adj + torch.eye(adj.size(0))
        self.D = torch.diag(torch.sum(adj, 1))
        self.D = self.D.inverse()
        self.A_hat = torch.mm(self.D, self.A_hat)
        self.W = nn.Parameter(torch.rand(input_channels, output_channels))

    def forward(self, H):
        out = torch.relu(torch.mm(torch.mm(self.A_hat, H), self.W))
        return out
```

```python
class Net1(torch.nn.Module):
    """
    Standard GCN model class
    """

    def __init__(self, adj, num_feat, num_hid, num_out):
        super().__init__()
        self.conv1 = GCNConv_Layer(adj, num_feat, num_hid)
        self.conv2 = GCNConv_Layer(adj, num_hid, num_out)

    def forward(self, H):
        H_next  = self.conv1(H)
        out = self.conv2(H_next)
        return out
```

Assembling both GCN layers for embedding

Two basic GCN layers for graph embedding

- ReLU as activation function
- $W_1.shape = [34, 10]$
- $W_2.shape = [10, 2]$
- The layers will progressively decrease the size of the feature vectors from size F, to size 10 and then finally size 2.

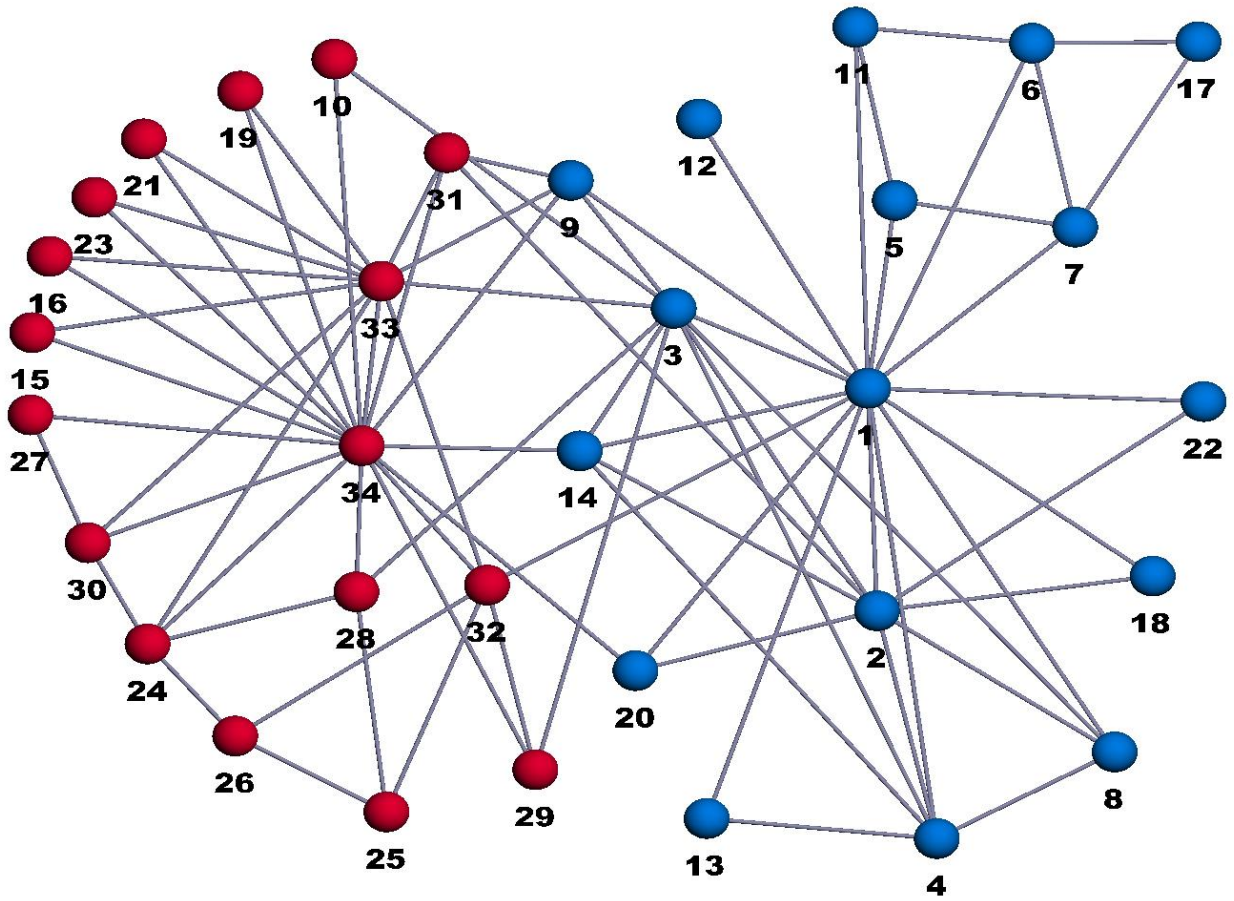
```
# No relevant features for nodes
# Using H = identity will make nodes features irrelevant
# and the model will have to learn from adjacency matrix only
H = torch.eye(A.size(0))
```

```
model = Net1(A, H.size(0), 10, 2)
criterion = torch.nn.CrossEntropyLoss(ignore_index = -1)
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum = 0.9)
loss = criterion(model(H), ground_truth)
```

Data splitting

- All nodes are in the training set.
- **Note:** We do not care about test/val for now, but in practice we should train/test/val split the nodes!

- Only Mr. Hi and Officer/Trainer nodes will already have their labels set correctly.



Trainer definition

Define a training function for our GCN + Logistic Regression model.

- **Loss:** Binary Cross Entropy
- **Optimizer:** SGD, with learning rate 0.01 and momentum 1.
- **Note:** no node features for now, making $H = I$, to make these nodes irrelevant in the forward propagation rule!

```
history1 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model(H), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model(H))

    if i%10 == 0:
        history1.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 0.9122539162635803
Cross Entropy Loss (iter = 10): = 0.7232621908187866
Cross Entropy Loss (iter = 20): = 0.6986600756645203
Cross Entropy Loss (iter = 30): = 0.6367791891098022
Cross Entropy Loss (iter = 40): = 0.5955305099487305
Cross Entropy Loss (iter = 50): = 0.547825813293457
Cross Entropy Loss (iter = 60): = 0.4989161193370819
Cross Entropy Loss (iter = 70): = 0.44563132524490356
Cross Entropy Loss (iter = 80): = 0.38999903202056885
Cross Entropy Loss (iter = 90): = 0.33411258459091187
Cross Entropy Loss (iter = 100): = 0.28105229139328003
Cross Entropy Loss (iter = 110): = 0.2333802878856659
Cross Entropy Loss (iter = 120): = 0.19256746768951416
Cross Entropy Loss (iter = 130): = 0.15904927253723145
Cross Entropy Loss (iter = 140): = 0.13240280747413635
Cross Entropy Loss (iter = 150): = 0.11144451797008514
Cross Entropy Loss (iter = 160): = 0.09498031437397003
```

Train model (no features)

- Overall, the model is learning but it takes time to do so properly.
- The model attempts to learn from the adjacency matrix and degree matrix only…
- It however has no useful nodes features it could rely on.

```
history1 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model(H), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model(H))

    if i%10 == 0:
        history1.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 0.9122539162635803
Cross Entropy Loss (iter = 10): = 0.7232621908187866
Cross Entropy Loss (iter = 20): = 0.6986600756645203
Cross Entropy Loss (iter = 30): = 0.6367791891098022
Cross Entropy Loss (iter = 40): = 0.5955305099487305
Cross Entropy Loss (iter = 50): = 0.547825813293457
Cross Entropy Loss (iter = 60): = 0.4989161193370819
Cross Entropy Loss (iter = 70): = 0.44563132524490356
Cross Entropy Loss (iter = 80): = 0.38999903202056885
Cross Entropy Loss (iter = 90): = 0.33411258459091187
Cross Entropy Loss (iter = 100): = 0.28105229139328003
Cross Entropy Loss (iter = 110): = 0.2333802878856659
Cross Entropy Loss (iter = 120): = 0.19256746768951416
Cross Entropy Loss (iter = 130): = 0.15904927253723145
Cross Entropy Loss (iter = 140): = 0.13240280747413635
Cross Entropy Loss (iter = 150): = 0.11144451797008514
Cross Entropy Loss (iter = 160): = 0.09498031437397003
```

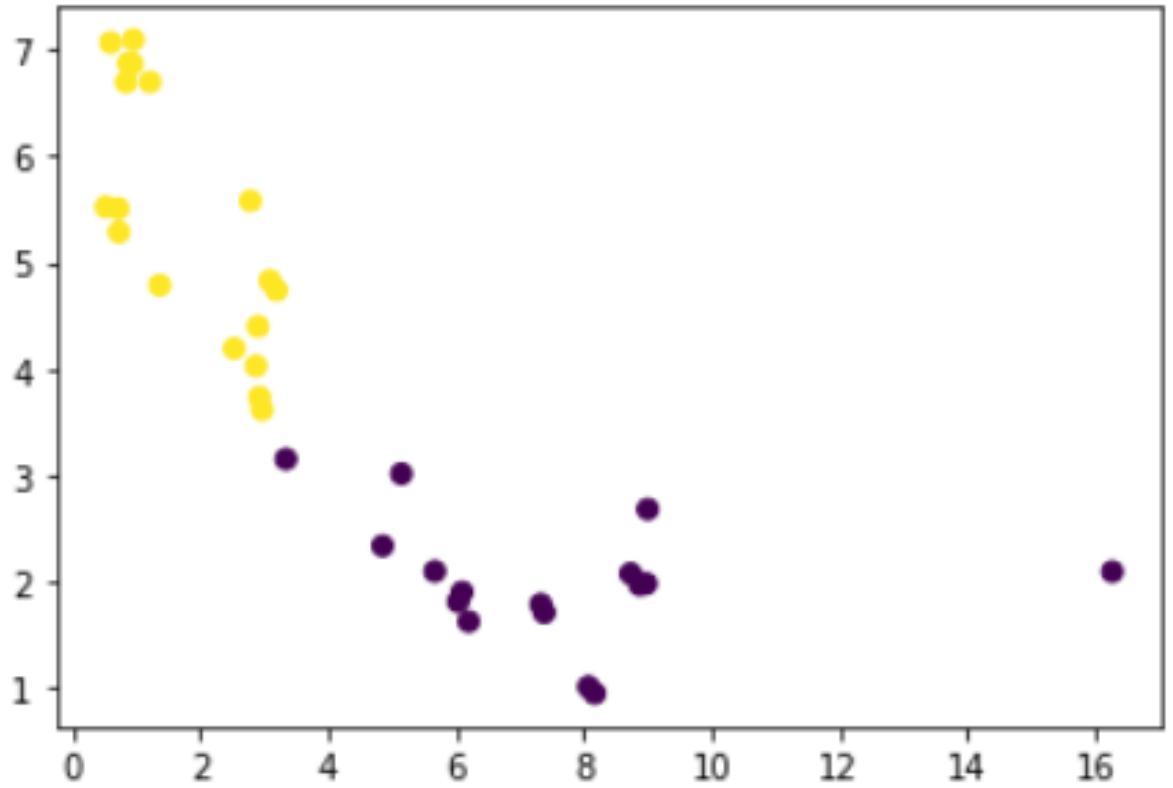
Train model (no features)

- Overall, the model is learning but it takes time to do so properly.

- The model attempts to learn from the adjacency matrix and degree matrix only…

- It however has no useful nodes features it could rely on.

```
plt.scatter(l.detach().numpy()[:,0], l.detach().numpy()[:,1], \
            c = ground_truth)
```



Building a Kipf GCN layer and model

- Similarly, let us create our own custom Kipf GCN layer block, using the PyTorch nn.Module class as superclass.

- Pass the adjacency matrix $\boldsymbol{A}$, compute $\hat{\mathrm{A}}$, $\widehat{\boldsymbol{D}}$, etc.
- Propagation rule:

$$\boldsymbol{H'} = \boldsymbol{\sigma}\left(\widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}\widehat{\boldsymbol{A}}\widehat{\boldsymbol{D}}^{-\frac{\mathbf{1}}{\mathbf{2}}}\boldsymbol{HW}\right)$$

```python
class GCNKipf_Layer(nn.Module):
    """
    Kipf GCN convolution layer class
    """

    def __init__(self, adj, input_channels, output_channels):
        super().__init__()
        self.A_hat = adj + torch.eye(adj.size(0))
        self.D = torch.diag(torch.sum(adj, 1))
        self.D = self.D.inverse().sqrt()
        self.A_hat = torch.mm(torch.mm(self.D, self.A_hat), self.D)
        self.W = nn.Parameter(torch.rand(input_channels, output_channels))

    def forward(self, H):
        out = torch.relu(torch.mm(torch.mm(self.A_hat, H), self.W))
        return out
```

```python
class Net2(torch.nn.Module):
    """
    Standard GCN model class
    """

    def __init__(self, adj, num_feat, num_hid, num_out):
        super().__init__()
        self.conv1 = GCNKipf_Layer(adj, num_feat, num_hid)
        self.conv2 = GCNKipf_Layer(adj, num_hid, num_out)

    def forward(self, H):
        H_next  = self.conv1(H)
        out = self.conv2(H_next)
        return out
```

As before…

Two Kipf GCN layers for graph embedding

- ReLU as activation function
- $W_1.shape = [34, 10]$
- $W_2.shape = [10, 2]$
- The layers will progressively decrease the size of the feature vectors from size F, to size 10 and then finally size 2.

```python
class GCNKipf_Layer(nn.Module):
    """
    Kipf GCN convolution layer class
    """

    def __init__(self, adj, input_channels, output_channels):
        super().__init__()
        self.A_hat = adj + torch.eye(adj.size(0))
        self.D = torch.diag(torch.sum(adj, 1))
        self.D = self.D.inverse().sqrt()
        self.A_hat = torch.mm(torch.mm(self.D, self.A_hat), self.D)
        self.W = nn.Parameter(torch.rand(input_channels, output_channels))

    def forward(self, H):
        out = torch.relu(torch.mm(torch.mm(self.A_hat, H), self.W))
        return out
```

```python
class Net2(torch.nn.Module):
    """
    Standard GCN model class
    """

    def __init__(self, adj, num_feat, num_hid, num_out):
        super().__init__()
        self.conv1 = GCNKipf_Layer(adj, num_feat, num_hid)
        self.conv2 = GCNKipf_Layer(adj, num_hid, num_out)

    def forward(self, H):
        H_next  = self.conv1(H)
        out = self.conv2(H_next)
        return out
```

As before…

Define a training function for our GCN + Logistic Regression model.

- **Loss:** Binary Cross Entropy
- **Optimizer:** SGD, with learning rate 0.01 and momentum 1.
- **Note:** no node features for now, making $H = I$, to make these nodes irrelevant in the forward propagation rule!

```
# No relevant features for nodes
# Using X = identity will make nodes features irrelevant
# and the model will have to learn from adjacency matrix only
H = torch.eye(A.size(0))
```

```
model2 = Net1(A, H.size(0), 10, 2)
criterion = torch.nn.CrossEntropyLoss(ignore_index = -1)
optimizer = optim.SGD(model2.parameters(), lr = 0.01, momentum = 0.9)
loss = criterion(model2(H), ground_truth)
```

Train model (no features)

- Overall, the model is learning but it takes time to do so properly.

- The model attempts to learn from the adjacency matrix and degree matrix only…
- It however has no useful nodes features it could rely on.
- Kipf formula seems to make the training slightly faster.

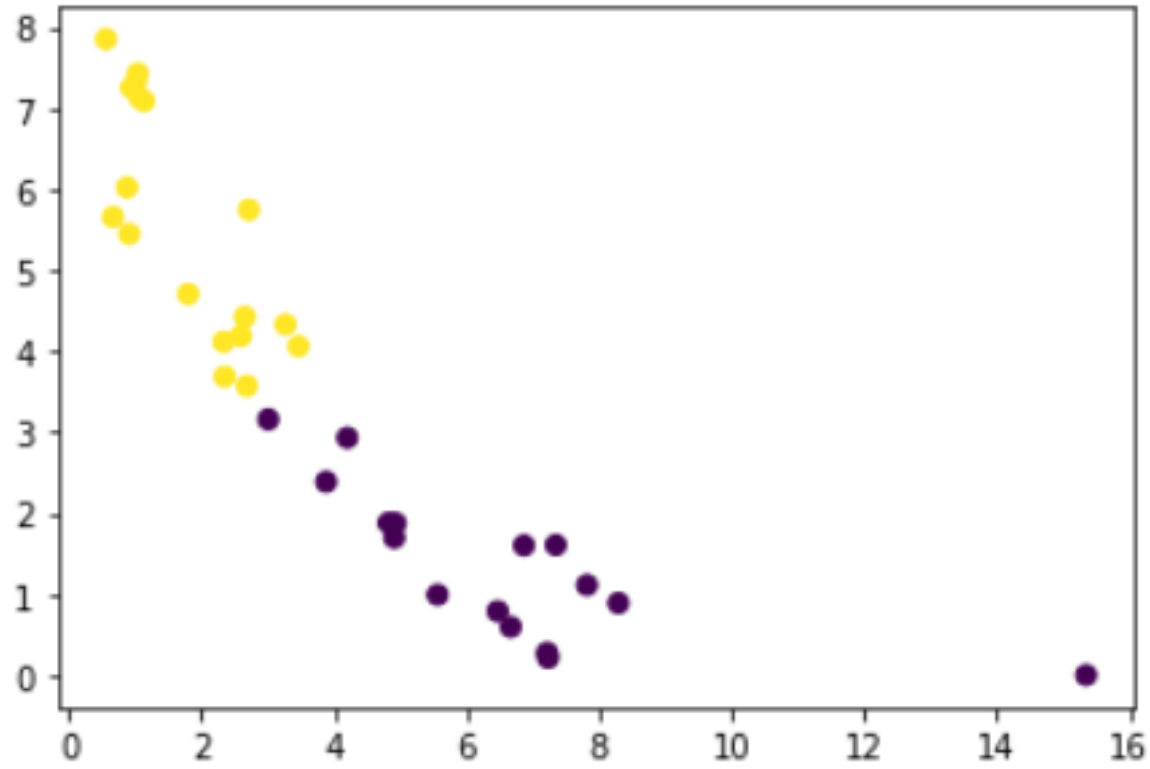
```
history2 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model2(H), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model2(H))

    if i%10 == 0:
        history2.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 0.7709766626358032
Cross Entropy Loss (iter = 10): = 0.6405860185623169
Cross Entropy Loss (iter = 20): = 0.6032090187072754
Cross Entropy Loss (iter = 30): = 0.537855327129364
Cross Entropy Loss (iter = 40): = 0.4822427034378052
Cross Entropy Loss (iter = 50): = 0.4223930835723877
Cross Entropy Loss (iter = 60): = 0.3637856841087341
Cross Entropy Loss (iter = 70): = 0.3073467016220093
Cross Entropy Loss (iter = 80): = 0.2565963864326477
Cross Entropy Loss (iter = 90): = 0.21271951496601105
Cross Entropy Loss (iter = 100): = 0.17605555057525635
Cross Entropy Loss (iter = 110): = 0.14626070857048035
Cross Entropy Loss (iter = 120): = 0.12245297431945801
Cross Entropy Loss (iter = 130): = 0.10356861352920532
```

Train model (no features)

- Overall, the model is learning but it takes time to do so properly.

- The model attempts to learn from the adjacency matrix and degree matrix only…
- It however has no useful nodes features it could rely on.
- Kipf formula seems to make the training slightly faster.



Conclusion

In this lecture

- Using graph types datasets
- Graph convolutions and graph embeddings
- How these convolutions relate to images
- Building a Graph Convolutional Neural Network for classification
- Training a Graph Convolutional Neural Network

In the next lectures

- (Feature Engineering for Graph Convolutional Neural Networks)
- Limits of basic approaches
- Graph Convolutional Neural Networks with Attention Mechanisms
- Some more advanced embeddings
- Open questions in Graph Convolutional Neural Networks

Learn more about these topics

Out of class, for those of you who are curious

- [TDS] A good article on Graph Theory and typical problems/algorithms with GIF visualizations! https://towardsdatascience.com/10-graph-algorithms-visually-explained-e57faa1336f3

- No extra readings for today
- (You already have many notebooks to look into!)

Maybe this will not be covered in Lecture 2

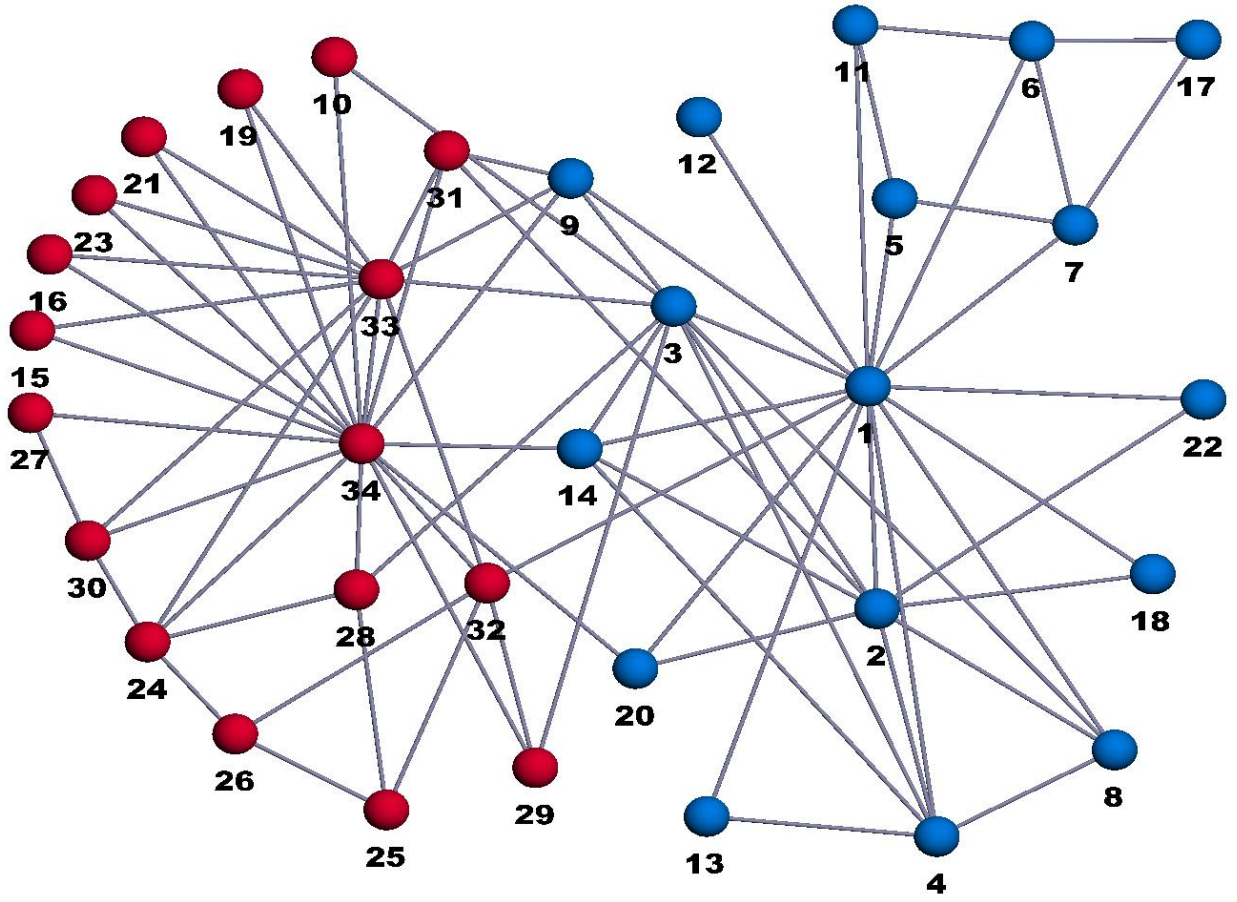
But it will be covered in Lecture 3 instead?

Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!



Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!

```python
def bfs(adj, start, goal):
    """
    Gives hop-distance between node start and node goal
    for given adjacency matrix.
    Returns zero if start = goal
    or goal not reachable from start.
    """

    if start == goal:
        return float(0)
    queue = [start]
    visited = []
    dist = float(0)

    while(len(queue) > 0):
        dist += 1
        temp = []
        for q in queue:
            neighbours_node = np.argwhere(adj[q]).reshape(1, -1)[0]
            if goal in neighbours_node:
                return dist
            else:
                for n in neighbours_node:
                    not_visited = (n not in visited)
                    not_queue = (n not in queue)
                    not_temp = (n not in temp)
                    if not_visited and not_queue and not_temp:
                        temp.append(n)
        visited.extend(queue)
        queue = temp
    return float(-1)
```

```python
y = bfs(adj, 1, 33)
print(y)
```

Build model: adding nodes features

- The nodes have no features.

→ Needs feature engineering

- Some interesting features for the nodes would be
  - the hop-distance to Mr. Hi (i.e. node 0),
  - and the hop-distance to the officer/trainer (i.e. node 33).
  - Use a simple BFS to calculate them!

```
# Adding relevant features (hop distance to nodes admin and instructor)
node_features = np.array([[bfs(adj, i, 0), bfs(adj, i, 33)] for i in range(34)])
H2 = torch.from_numpy(node_features).float()
print(H2)
```

```
tensor([[0., 2.],
        [1., 2.],
        [1., 2.],
        [1., 2.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 1.],
        [2., 1.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 3.],
        [1., 1.],
        [3., 1.],
        [3., 1.],
        [2., 4.],
        [1., 3.],
        [3., 1.],
        [1., 1.],
        [3., 1.],
        [1., 3.],
        [3., 1.],
        [3., 1.],
        [2., 2.],
        [2., 2.],
        [3., 1.],
        [2., 1.],
        [2., 1.],
        [3., 1.],
        [2., 1.],
        [1., 1.],
        [2., 1.],
        [2., 0.]])
```

Trainer definition

Define a training function for our GCN + Logistic Regression model.

- **Loss:** Binary Cross Entropy
- **Optimizer:** SGD, with learning rate 0.01 and momentum 1.

- **Note:** let us now try with these relevant node features H_2!

Train model (with some features)

- Overall, the model is now able to train properly.
- The model attempts to learn from the adjacency matrix and degree matrix only.
- The hop-distances carry an information on the proximity of members to Mr. Hi and the Officer/Trainer.

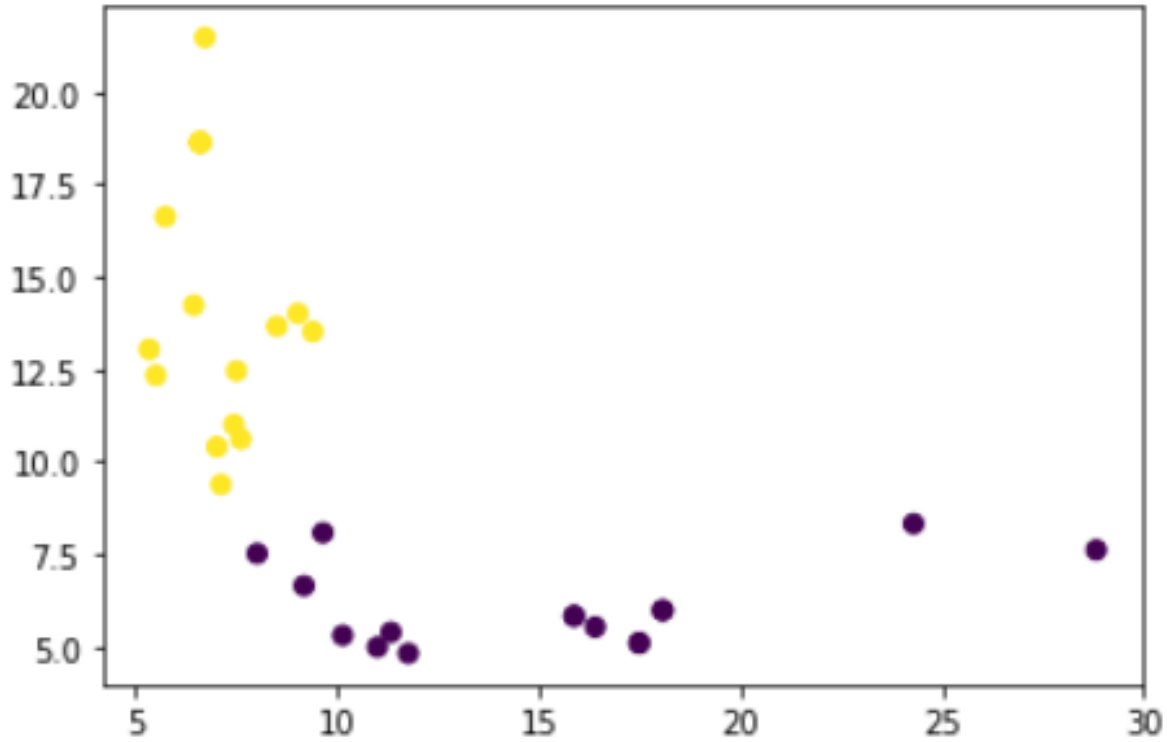
```
model3 = Net2(A, H2.size(1), 10, 2)
criterion = torch.nn.CrossEntropyLoss(ignore_index = -1)
optimizer = optim.SGD(model3.parameters(), lr = 0.01, momentum = 0.9)
loss = criterion(model3(H2), ground_truth)
```

```
history3 = []
for i in range(500):
    # Forward pass
    optimizer.zero_grad()
    loss = criterion(model3(H2), current)

    # Backprop
    loss.backward()
    optimizer.step()

    # For display later
    l = (model3(H2))

    if i%10 == 0:
        history3.append(loss.item())
        print("Cross Entropy Loss (iter = {}): =".format(i), loss.item())
```

```
Cross Entropy Loss (iter = 0): = 1.4348474740982056
Cross Entropy Loss (iter = 10): = 0.07234100997447968
Cross Entropy Loss (iter = 20): = 0.005204013083130121
Cross Entropy Loss (iter = 30): = 0.0015288800932466984
Cross Entropy Loss (iter = 40): = 0.0008693403797224164
Cross Entropy Loss (iter = 50): = 0.0007051686989143491
Cross Entropy Loss (iter = 60): = 0.000646679662168026
Cross Entropy Loss (iter = 70): = 0.0006161221535876393
```

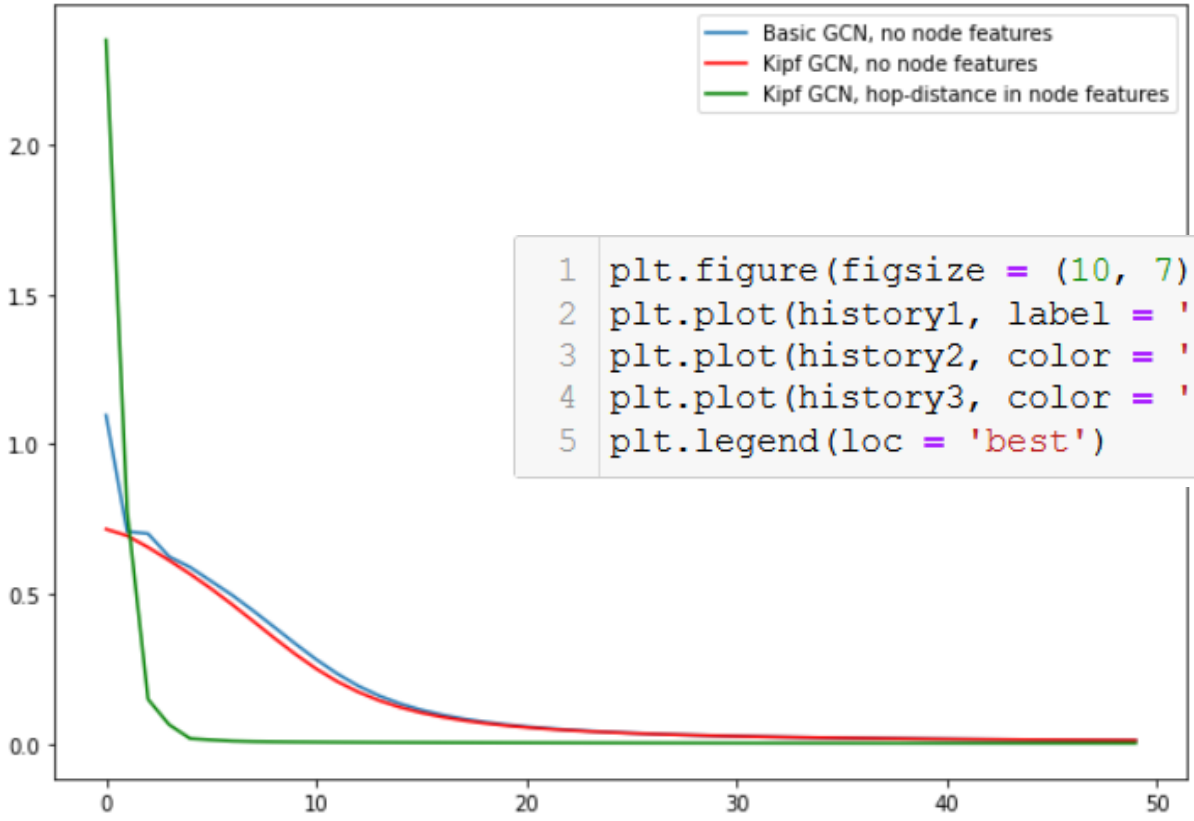
Train model (with some features)

- Overall, the model is now able to train properly.
- The model attempts to learn from the adjacency matrix and degree matrix only.
- The hop-distances carry an information on the proximity of members to Mr. Hi and the Officer/Trainer.



Train model (with some features)

Important message: Additional nodes features make the training much more efficient! Features engineering is key with graphs!




```
plt.figure(figsize = (10, 7))
plt.plot(history1, label = 'Basic GCN, no node features')
plt.plot(history2, color = 'red', label = 'Kipf GCN, no node features')
plt.plot(history3, color = 'green', label = 'Kipf GCN, hop-distance in node features')
plt.legend(loc = 'best')
```

Additional features?

We could think of additional features to help the Graph Convolutional Neural Network learn. For instance:

- Additional node features: for instance, number of close friends to Mr. Hi and Officer/Trainer among the neighbors of the node (close friends = nodes with hop-distance of 1).
- Edge features: for instance, add a random “Friendship” value to the edges, that quantifies friendship between two nodes.
  Open question: Later on, how would you modify your propagation formulas to take into account the edge weights?

Challenge: Leaving the implementation of these extra features as challenge for you to practice!

Why did we use Laplacians for our spectral embedding layers? (intuition, out of class)

- **Embedding as a minimization problem:** a "good" embedding
  $\forall i \in [1, N], p_i \colon h_i \in \mathbb{R}^F \rightarrow v_i \in \mathbb{R}^{F'}$

Encodes nodes features h_i into vectors v_i, such that connected nodes stay as close a possible.

A "good" embedding matrix

$$\mathrm{P} = [p_1(h_1), \ldots, p_N(h_N)]$$

is obtained through the following minimization

$$\min_P \left[\sum_{i=1}^{N} \sum_{j=1}^{N} \left(p_i(h_i) - p_j(h_j) \right)^2 \right]$$

Which is equivalent to

$$\min_P [P^T L P],$$
$$with\ P^T P = I\ and\ P^T I = 0$$

Why did we use Laplacians for our spectral embedding layers? (intuition, out of class)

- **Embedding as a minimization problem:** a "good" embedding $\forall i \in [1, N], p_i\colon h_i \in \mathbb{R}^F \rightarrow v_i \in \mathbb{R}^{F'}$

Encodes nodes features h_i into vectors v_i, such that connected nodes stay as close a possible.

This embedding is solution to

$$\min_P [P^T L P],$$
$$with\ P^T P = I\ and\ P^T I = 0$$

- **Solution:** The solution is provided by the matrix of eigenvectors corresponding to the F' lowest non-zero eigenvalues of the eigenvalue problem

$$LP = \lambda P$$

Why did we use Laplacians for our spectral embedding layers? (intuition, out of class)

- Using the random walk Laplacian L_{rw}, instead of the "normal" Laplacian adds a **normalization** factor to the embedding.
- Using the symmetric Laplacian L_{sym}, instead of the "normal" Laplacian adds a **normalization factor + accounts for neighboring nodes in the embedding.**

- If you are curious, find more details (mathematical foundations behind embedding problems, proofs, applications, etc.)
- UC San Diego lecture (more specifically, slides 14-end): https://cse291-i.github.io/WI18/LectureSlides/L14_GraphLaplacian.pdf