Metrics agnostic to classification threshold:

- ROC AUC

Setting Up GNN Prediction Tasks

Now, that we have talked about the pipeline, the ~~subtle~~ very important question comes,

How do we split our dataset into train/validation/test set?

Dataset split
- Fixed Split
- Random Split.

A Fixed Split: We will split our dataset only once in a fixed split. The split will be into three parts:

- Training Set: used for optimising GNN parameters
- Validation Set: develop model / hyper parameters
- Test Set: To evaluate the model on unseen data.

However, the problem is sometimes the test set might have information leakage from train and validation set. This solely happens due to the dataset being a graph.

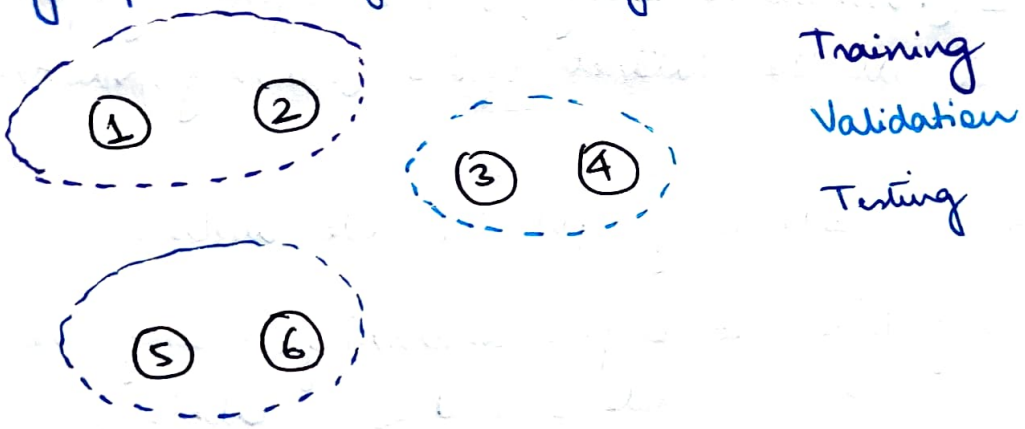
So, the test cannot be guaranteed to be truly unseen and so this is a problem.

Random Split : In this the original data is split into randomly into train, validation and ~~set~~ test set over different seeds and then average performance ~~of~~ over these splits is reported.

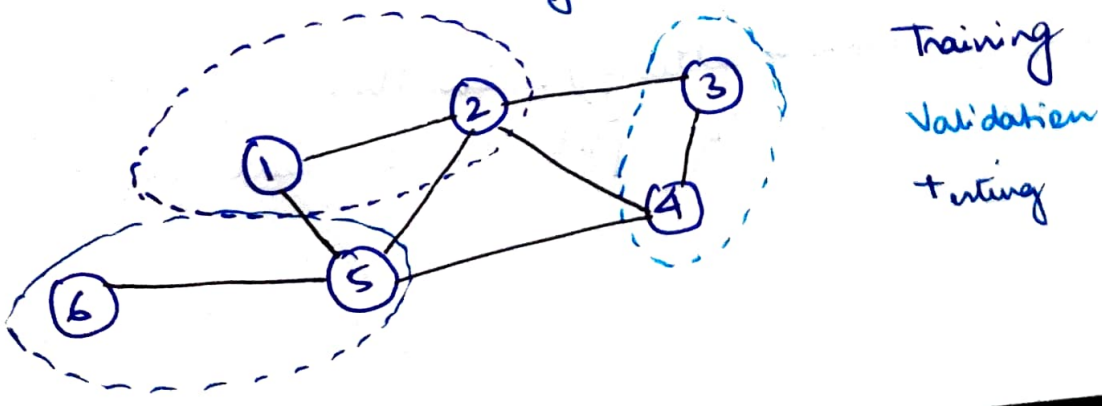
This creates more robustness in our splits and helps the models generalize well.

Why Splitting Graphs is Special?

Suppose we have an image / document dataset. We assume that each data point is independent and so, they can be easily split without any problem of data leakage.



However, ~~the~~ splitting graph is different ~~and~~ as each datapoint (nodes/edges) are not independent of each other. For e.g ~~in~~ consider the task of node classification ~~and~~ where each data point is a node. Assume the graph looks like as depicted below:



It is clear from the figure that node 1 and node 5 are in different splits. However, node 5 will participate in node 1's message passing and thus will affect node 1's prediction. which is a problem.

So, in graphs Data points are not ~~independents~~ independent which makes them ~~different~~ difficult to split.

Then, what are our options?

Solution 1 (Transductive Setting): The input graph can be observed in all the dataset splits (training, validation and test set).

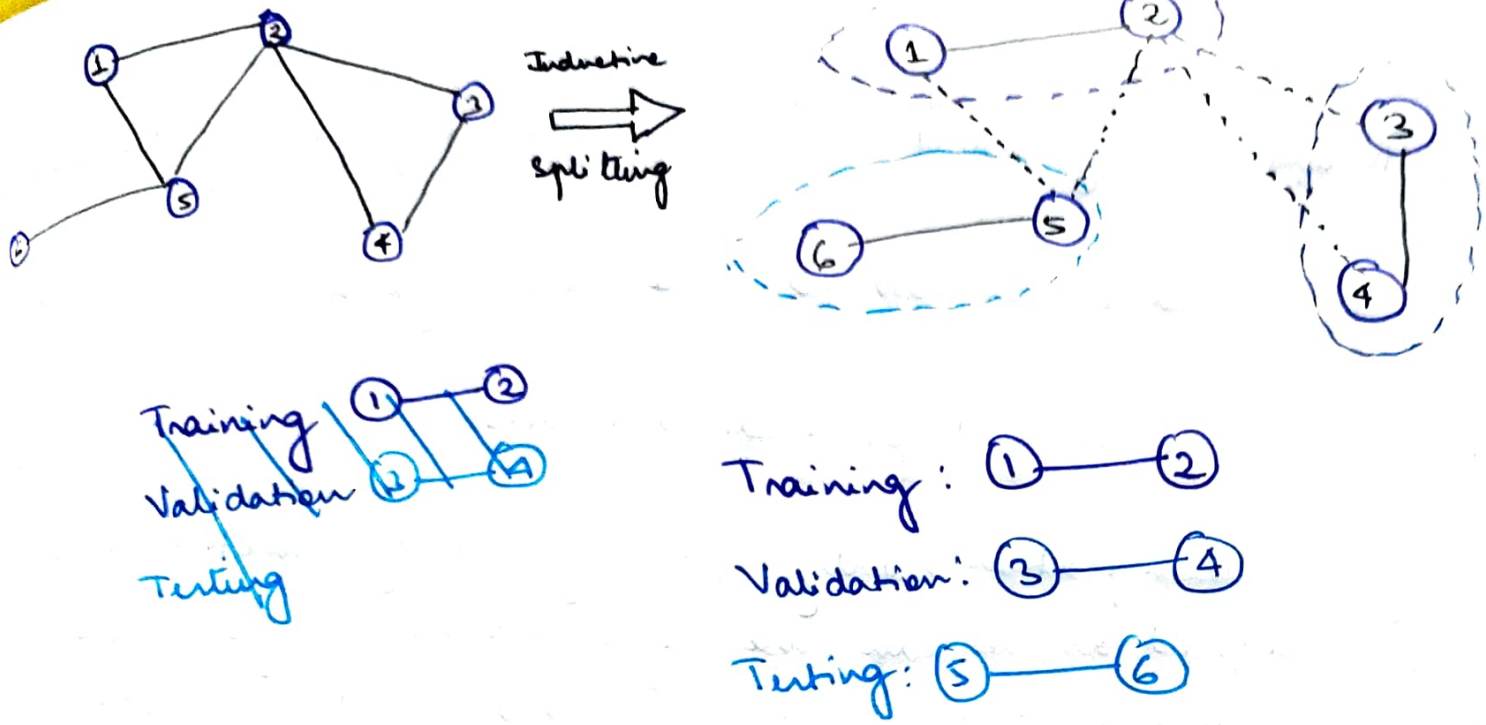
i.e ~~at to~~ we will only split the node labels.

At training time, we compute embeddings using the entire graph and train using only node 1 and 2's labels.

~~At~~ validation time, we again compute embeddings using the entire graph and tune the hyperparameters using node 3 and 4's labels.

~~At~~ testing time, we test using node 3 and 4's labels.

Solution 2 (Inductive Setting): In this approach, we break the edges between splits to split the original graph into multiple graphs.



So, now we have 3 independent subgraphs which can be leveraged to train, validate and test the models.

Since the 3 graphs are independent, node 5 will no longer affect node 1's prediction.

- At training time, we compute embeddings using the graph over node 1 and 2 and train using node 1 and 2's labels.

- At validation time, we compute embeddings using the graph over 3 and 4, and validate using node 3 and 4's labels.

- At testing time, we test using node 5 and node 6's labels.

However, due to breaking the graph, some of the edges might are thrown away which results in loss of structural information.

So, This method is not advised for a small Graphs.

These methods are disamed for Node classification.

For Graph classification, only splitting using Inductive Setting is well defined as we can use some independent graphs set to train, some independent graphs to validate and some independent graphs to test the model.

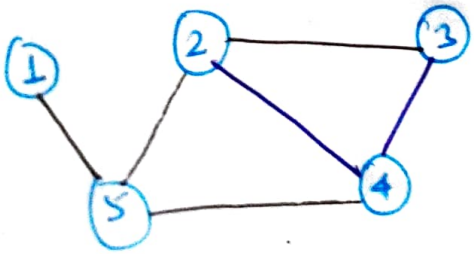
However, the trickiest of the three tasks is Link Prediction.

Setting up Link Prediction

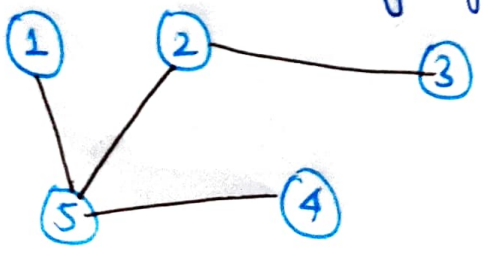
The goal of link prediction is to predict missing edges -

Since, link prediction is a type of self supervised learning where we have to create labels and dataset splits of our own, setting up link prediction can be tricky.

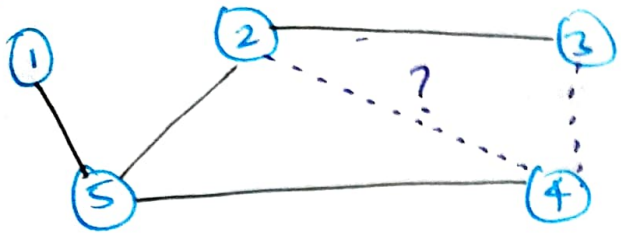
Concretely, we we need to hide some edges from the GNN and let it the GNN predict the missing edges.



Original Graph



Input graph to GNN.



Predictions made by the GNN.

The first step to set up link prediction task is to divide the edges of the original graph into two types.

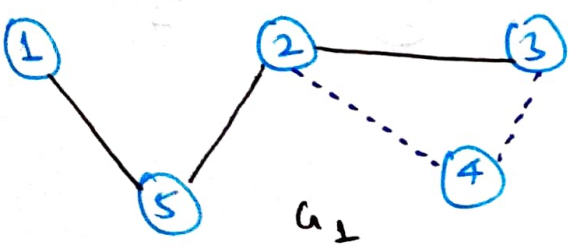
Edges → Message passing edges - Used to learn embeddings
→ Supervision edges - Used as supervision to train the GNN.

Once we're done with fixing the message passing and supervision edges, we'll feed only the message passing edges into the GNN and not the supervision edges as these edges will be used for training.

The next step is to split edges into train/validation/test edges

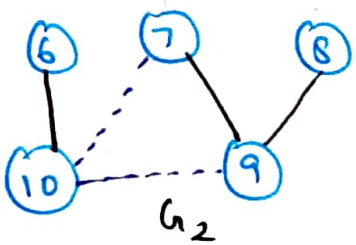
Inductive link prediction split :

~~For this~~ Suppose we have a dataset of 3 graphs. ~~Each inductive~~ These three graphs will be independent and serve as training, validation and test graphs.



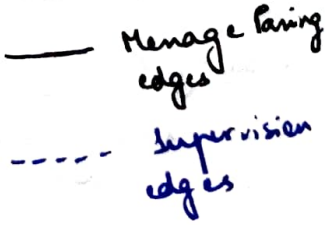
G_1

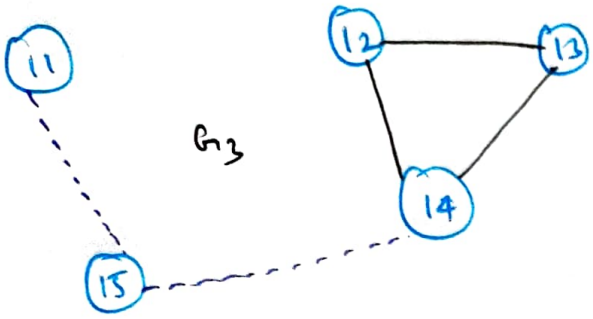
Training Set



G_2

Validation Set



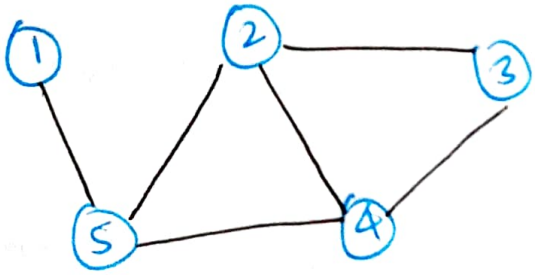


Transductive link prediction split : In this we have only one graph, and we observe the entire graph in all dataset splits.

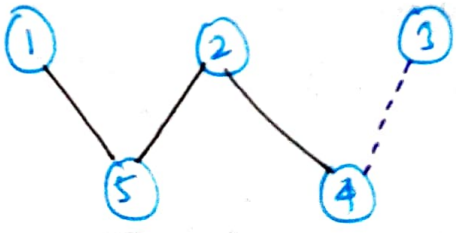
- But since edges are both part of graph structure and supervision, we need to hold out validation / test edges

- To train the training set, we further need to hold out supervision edges for the training set.

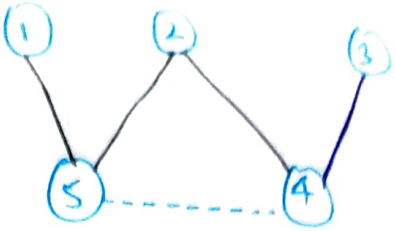
Suppose we have the original graph.



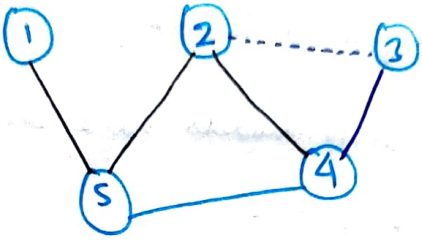
(1) At training time we use training message edges to predict the training supervision edges.



(2) At validation time, we use the training message edges ~~to predict~~ and training supervision edges to predict validation edges.



(3) At test time we use training message edges, training supervision edges, and validation edges to predict the test edges.



So, the transductive setting can be thought of as the graph growing over time.

And splitting it is basically splitting it over three time intervals.

After training, the supervision edges are known to the GNN. Therefore, an ideal model should use supervision edges in message passing at validation time. The same applies to the test time.

How Expressive are Graph Neural Networks?

In this part, we are shifting gears to understand the theory of Graph Neural Networks inorder to answer the question:

How powerful are GNNs?

Many GNN models have been proposed such as GCN, GAT, GraphSAGE, etc.

So, out of these we try to understand what is the most expressive GNN?

We also try to answer what can GNNs learn and not learn?

Also, How to design a maximally expressive GNN model?

From what we've studied so far, different GNN models use different neural networks in the box.

GCN (mean-pool)

For GCN as we've studied, we use

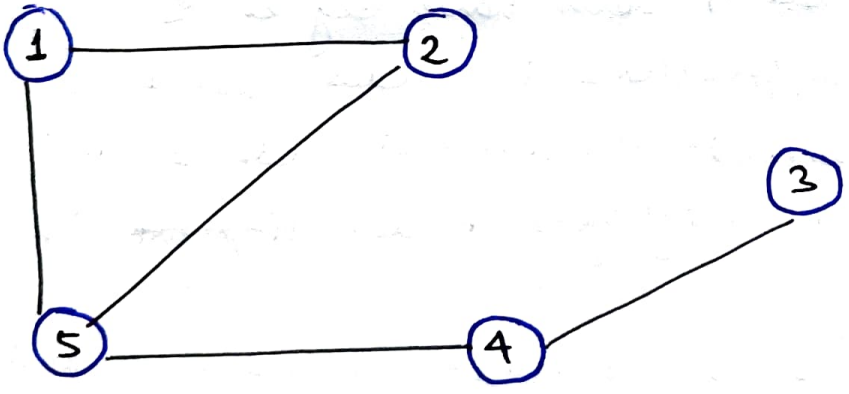
Element wise mean pool + Linear Transformation + ReLU non linearity

Graph SAGE

In graph SAGE, we can choose what aggregator to use. Consider we are using the MaxPooling function as the aggregator. Then the ~~so~~ neural network of graph SAGE consists of:

MLP + Element wise MaxPooling

In order to understand how expressive and powerful our Graph Neural Network models actually are let us consider a graph where every node have the same feature. Nodes with same features are colored with the same color.



○ → ■ features

Clearly, all the feature vectors are the same.

Now, the key question is:

How well can a GNN distinguish different graph structures?

i.e How well can a GNN distinguish two different nodes.

Before getting to answer the question, we make a point that in GNNs we're specifically interested in the local neighborhood structure around each node.

On the basis of these local neighborhood structures, we can say that a GNN can differentiate nodes due to the fact that GNNs only considers these local neighborhood structures to learn embeddings.

Consider the graph given in the previous page.

Clearly node 1 and 5 have different local neighborhood structures because they have different node degrees.

However node 1 and node 4 both have degree 2. So, capturing only the degree to differentiate the nodes would not work. Although the neighborhood structures of node 1 and 4 are different because their neighbors have different node degrees.

Node 1 has neighbors of degree 2 and 3

Node 4 has neighbors of degree 1 and 3

So, they have different neighborhood structures.

Now, lets look at node 1 and 2. Since, they are symmetric in the graph they are indistinguishable.

- They have the same degree of 2
- Both of them have neighbors having degree 2 and 3.

Even if we go to a step deeper to consider the 2 hop neighborhood the 2 hop neighbors of node 1 and 2 also have the same degree.

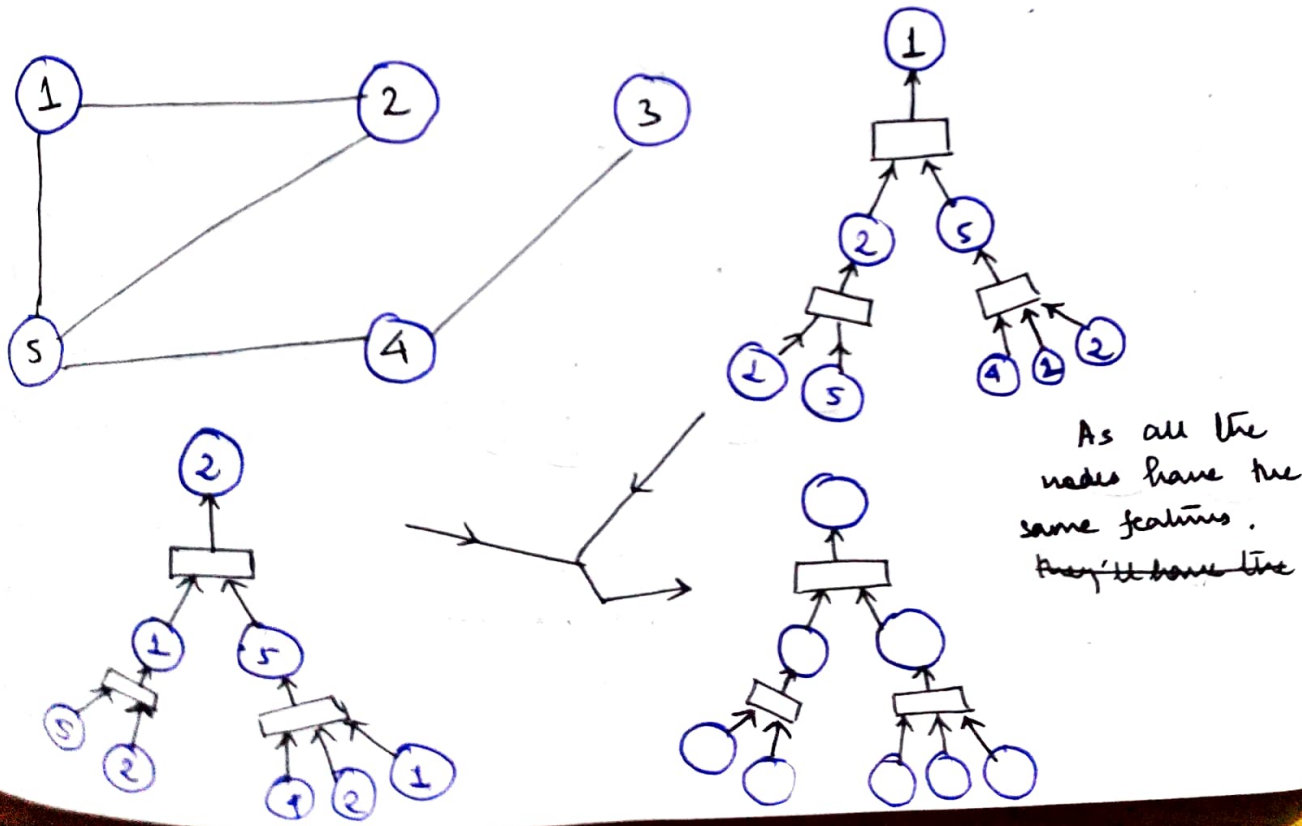
Thus as every node has the same features, and as the local neighborhood structures of ~~both~~ node 1 and 2 are exactly the same, ~~so~~ GNNs will not be able distinguish these ~~no~~ two nodes.

So, we are trying to understand

Can GNN node embeddings distinguish different node's local neighborhood structures? If so, when? If not, when will a GNN fail?

We'll address the question using the key concept of computational graph.

We know a GNN generates node embeddings through a computational graph defined by the neighborhood.



Clearly, from the computational graph, both node 1 and node 2 will have the same computational graph.

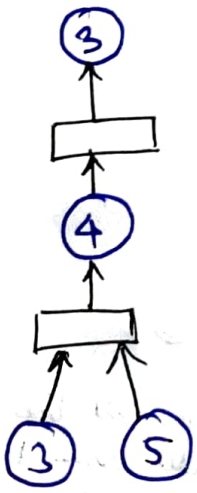
Since, node 1 and node 2 have the same computational graph, and since GNN doesn't care about node id's, ~~the~~ and ~~f~~ as all the nodes have the same features

GNNs WON'T BE ~~DISTIN~~ ABLE TO DISTINGUISH NODES 1 and 2.

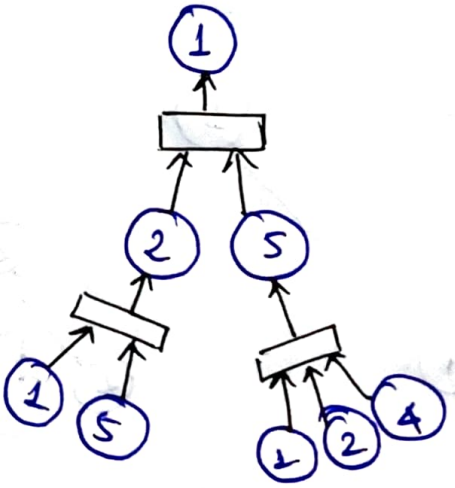
Thus, A GNN will generate the same embedding for nodes 1 and 2 because:

- Their computation graphs are the same
- They have the same node features

~~for~~ Now, consider the computation graphs of node 1 and 3.



Node 3's Computational graph



Node 1's Computational graph.

Clearly node 1 and node 3 have different computational graph and so, the GNN will learn different node embeddings for node 1 and node 3.

Thus, in general

As different local neighborhoods define different computational graphs, a GNN should learn different embeddings for different local neighborhoods.

But, does a GNN learn different embeddings for different neighborhood structures?

This is the key question which we are trying to answer.

- GNN's node embeddings capture rooted subtree structures.
- Most expressive GNN maps different rooted subtrees into different node embeddings.

Thus, while design GNNs we should ensure if two nodes have different computation graphs, then the GNN learn two different embeddings for them.

Recall,

A function $f: X \to Y$ is injective if

$$f(a) = f(b)$$
$$\Rightarrow a = b \quad \text{for all } a, b \in X.$$

Most expressive GNN should map subtrees to the node embeddings injectively.

i.e different subtrees is mapped to different embeddings and same subtrees will be mapped to the same embeddings.

So, we want to develop a Graph Neural Network (GNN) that maps local neighborhood structures to their corresponding embeddings injectively. Since, we use neighborhood aggregation function at each step the most expressive GNN will have an injective neighborhood aggregation at every step.

Designing the most powerful Graph Neural Network

Now, we have all the information ~~to~~ required to design the most powerful neural network out there.

The major ~~takeaway~~ takeaway from the previous part is that expressive power of GNNs can be characterized by that of neighborhood aggregation function they use.

Injective aggregation function leads to the most expressive GNNs.

Now, let us analyse the expressive powers of the ~~GNN~~ existing GNNs.

Observation: A neighborhood aggregation function of a GNN can be abstracted as a function over a multiset (a set with repeating elements).

$\{x, x, y, z, a, a\}$ is an example of a multiset.

~~• GCN (mean pool)~~

• GCN (mean-pool)

It uses element wise mean pooling over neighboring node features.

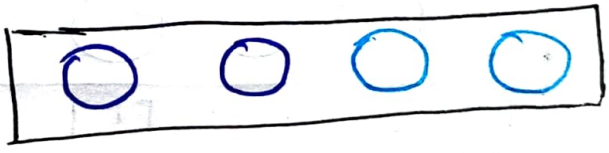
$$\text{Mean}(\{x_u\}_{u \in N(v)})$$

Is element wise mean pooling injective?

No, element wise mean pooling is not an injective function. The failure case is:

Consider a neighborhood

~~and~~ and another neighborhood



O - one feature

O - another feature

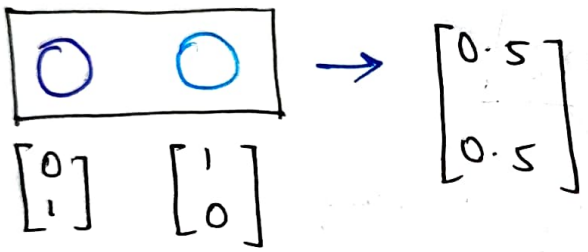
Clearly, in both of the neighborhoods the mean will be the same, which is a failure case as the two neighborhoods are different.

So, for GCN, we have the following theorem:

GCN's aggregation function cannot distinguish different multi-sets with the same color proportion.

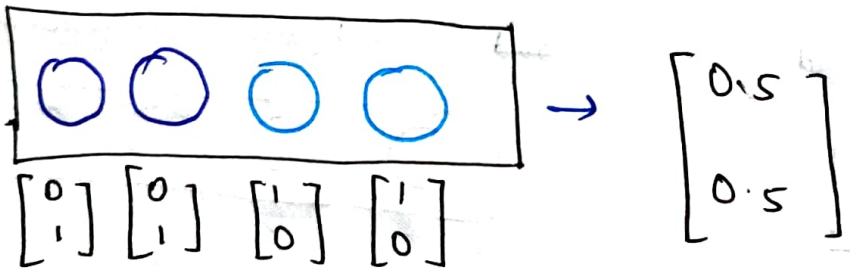
To illustrate the previous example, assume

Then, GCN of



which is exactly the same!

and, GCN of



GraphSAGE (max pool)

In the max pooled GraphSAGE, we apply an MLP and take an element wise max.

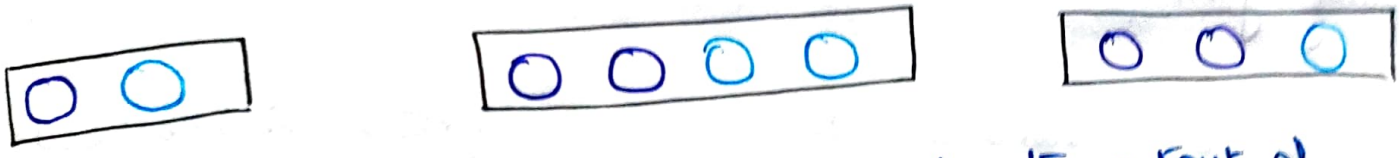
Is element wise max pooling injective?

The answer is No.

Theorem: Graph SAGE's aggregation function cannot distinguish different multisets with the same set of distinct colors.

Failure Case:

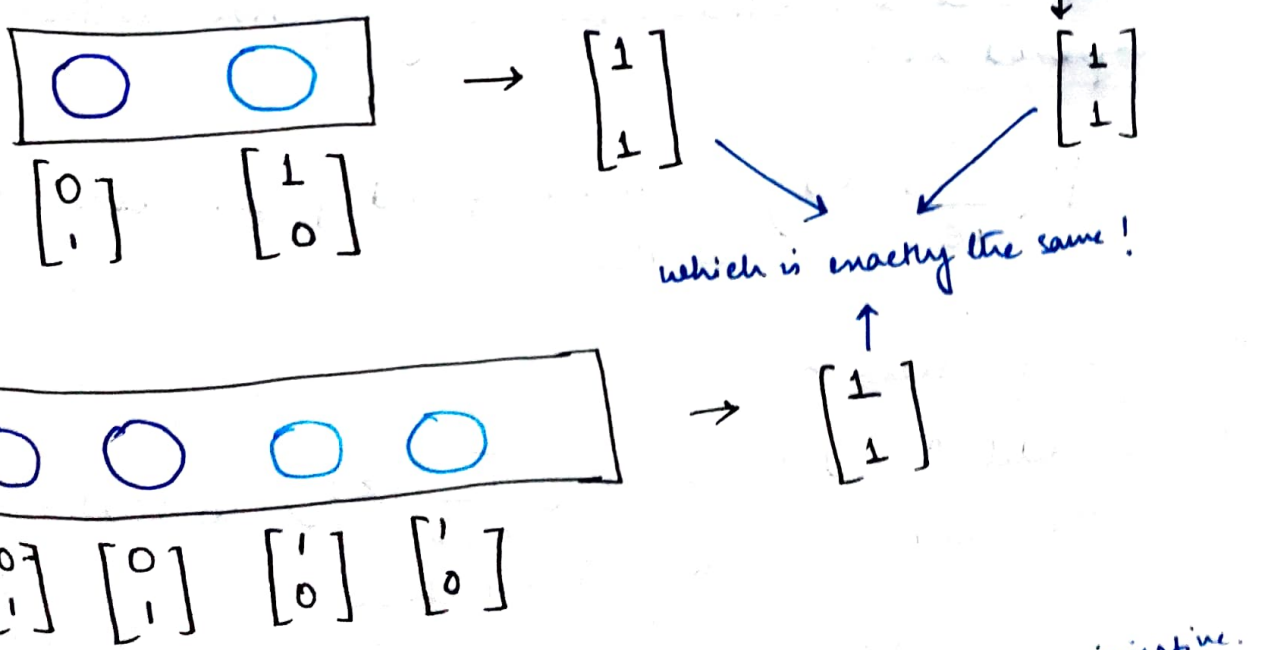
Consider the following neighborhoods



In all of the above described neighborhoods, the output of the aggregator will be the same.

To illustrate consider ○ → $\begin{bmatrix}0\\1\end{bmatrix}$ and ○ → $\begin{bmatrix}1\\0\end{bmatrix}$

Then, MaxPool of:



Thus, Graph SAGE's aggregation function is also not injective.

∴ GCN and GraphSAGE are not maximally powerful GNNs.

But, our goal is :

Design maximally powerful GNNs in the class of message passing GNNs.

This can be achieved by designing an injective neighborhood aggregation function over multisets.

So, next we design a neural network that can model injective multiset function.

Inorder to ~~def~~ design an injective multiset function, we state the following theorem:

Theorem: Let S be a multiset. Then any injective function over S can be expressed as :

$$\Phi\left(\sum_{x \in S} f(x)\right)$$ where Φ and f are some non linear functions.

Now, the question is :

How to model Φ and f in $\Phi\left(\sum_{x \in S} f(x)\right)$ so that ~~it is~~ the resulting function is injective.

We use a Multilayered Perceptron.

~~From Universal~~

Theorem: Universal Approximation Theorem

1 hidden layer MLP with sufficiently large hidden dimensionality and appropriate non linearity $\sigma(\cdot)$ (including ReLU and sigmoid) can approximate any continuous function to an arbitrary accuracy.

Based on this we ~~can state~~ can state that using two MLP's will let us ~~approx~~ learn ϕ and f such that $\phi\left(\sum_{x \in S} f(x)\right)$ is injective.

Thus, we have arrived at a neural network that can model any injective function.

$$\text{MLP}_{\phi}\left(\sum_{x \in S} \text{MLP}_{f}(x)\right)$$

In practice, MLP hidden dimensionality of 100 to 500 is sufficient.

~~Hmm,~~

Graph Isomorphism Network (GIN) comprises of the above neighborhood aggregation function and hence is injective!

~~GNN is the most expressive~~

GIN is the most expressive GNN in the class of message passing GNNs.

Now, we describe the full model GIN relating it to WL graph Kernel.

We'll see how GIN is a neural network version of WL graph kernel.

The Weisfeiler Lehman (WL) graph kernel is graph kernel method used for comparing the structural similarity of graphs.

It tells whether two graphs are isomorphic or not.

The WL graph kernel is based on a color refinement algorithm that iteratively assigns color label to each node.

The algorithm states

- Assign an initial color of $c^{(0)}(v)$ to each node v.

Iteratively refine node colors by:

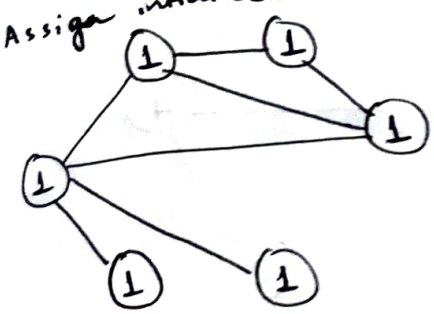
$$c^{(k+1)}(v) = \text{HASH}\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$$

where HASH maps different inputs to different colors

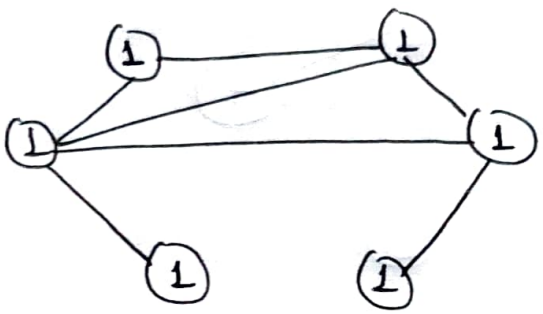
After K steps of color refinement, $c^{(K)}(v)$ summarizes the structure of K-hop neighborhood.

Consider the two graphs:

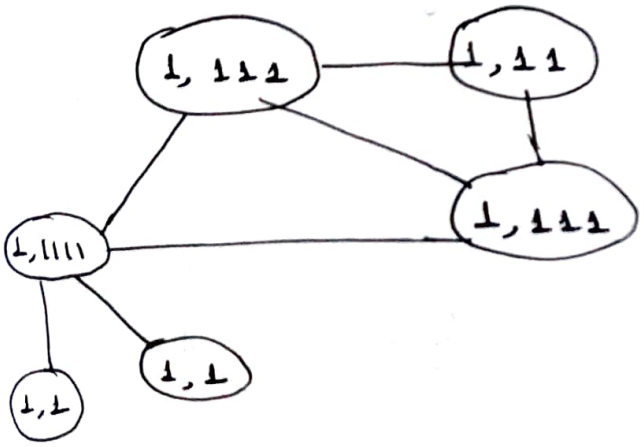
Assign initial colors

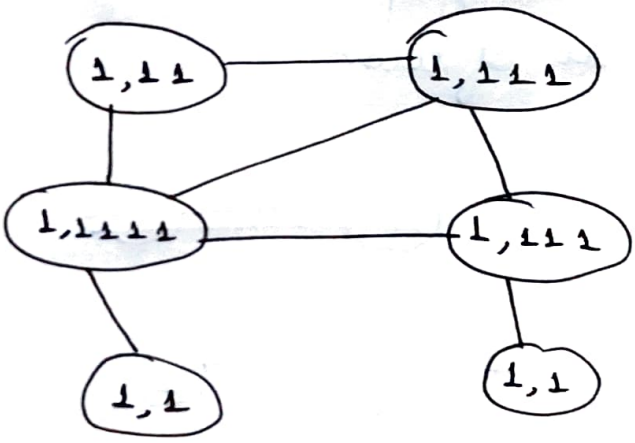


$c^0(v) = 1 \quad \forall v.$



Aggregate neighboring colors





Now let us hash the aggregate color labels into new labels. Let us assume our ~~de~~ HASH function is injective:

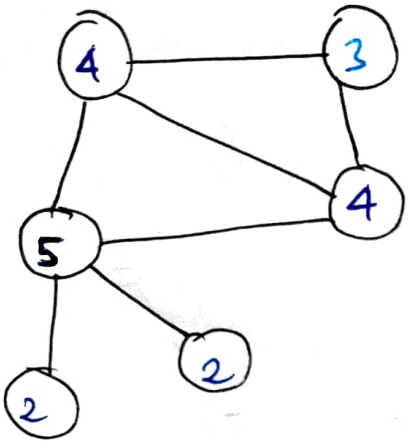
1, 1 ---> 2

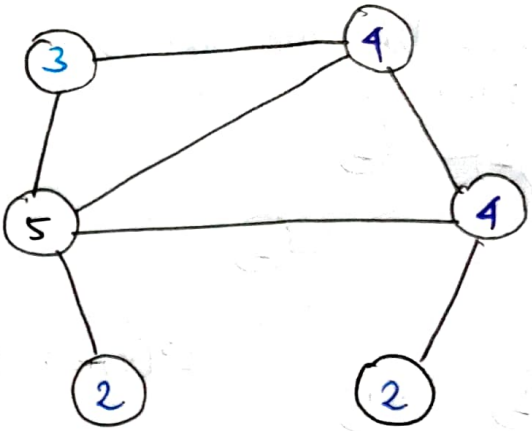
1, 11 ---> 3

1, 111 ---> 4

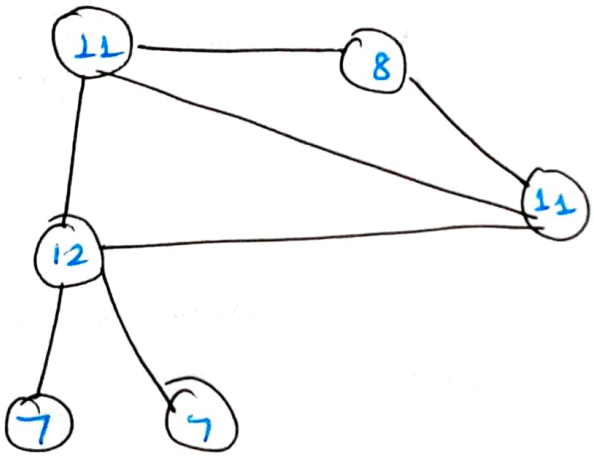
1, 1111 ---> 5

Then the graph becomes:

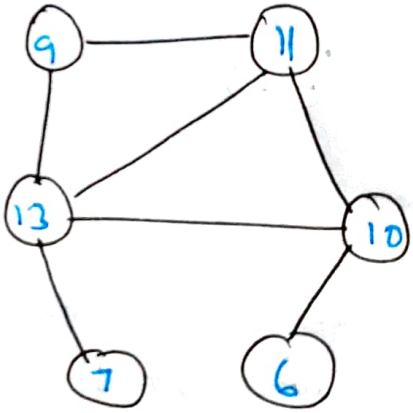




- This process continues until a stable coloring is reached
- Two graphs are considered isomorphic if they have a same set of colors.



$\neq$



From the hash function $\left(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}\right)$ it has a correspondence to aggregation functions of GNN; only in that case we use a neural network instead of a hash function

Now, coming back to GIN a theorem states:

~~Any injective function over the tuple~~

$$\left(h^{(k)}(v), \sum_{u \in N(v)} h^{(k)}(u)\right)$$

Any injective function over the tuple $\left(h^{(k)}(v), \{h^{(k)}(u)\}_{u \in N(v)}\right)$ can be modeled as:

$$\text{MLP}_\Phi\left((1+\varepsilon)\cdot \text{MLP}_f\left(h^{(k)}(v)\right) + \sum_{u \in N(v)} \text{MLP}_f\left(h^{(k)}(u)\right)\right)$$

where ε is a learnable scalar.

The Complete GIN Model

Given: A graph G with a set of nodes V.

- Assign an initial vector $h^{(0)}(v)$ for every $v \in V$.
- Iteratively update the nodes as:

$$h^{(k+1)}(v) = \text{MLP}_\Phi\left((1+\varepsilon)\text{MLP}_f\left(h^{(k)}(v)\right) + \sum_{u \in N(v)} \text{MLP}_f\left(h^{(k)}(u)\right)\right)$$

where, ε is a learnable scalar.

- After K steps of GIN iterations, $h^{(K)}(v)$ summarizes the structure of K hop neighborhood.

GIN can be understood as differentiable neural version of WL Graph Kernel.

	Update target	Update function
WL Graph Kernel	Node colors (one-hot)	HASH
GIN	Node embeddings (low-dim)	GINConv

Advantages of GIN over WL graph Kernels:

- Node embeddings are low dimensional and hence, they can capture fine grained similarity of different nodes.
- Parameters of the update function can be learned for the downstream tasks.

How Powerful is GIN?

- WL Graph Kernel has been both theoretically and empirically shown to distinguish most of the real world graphs.
- Hence, GIN is also powerful enough to distinguish most of the real world graphs.