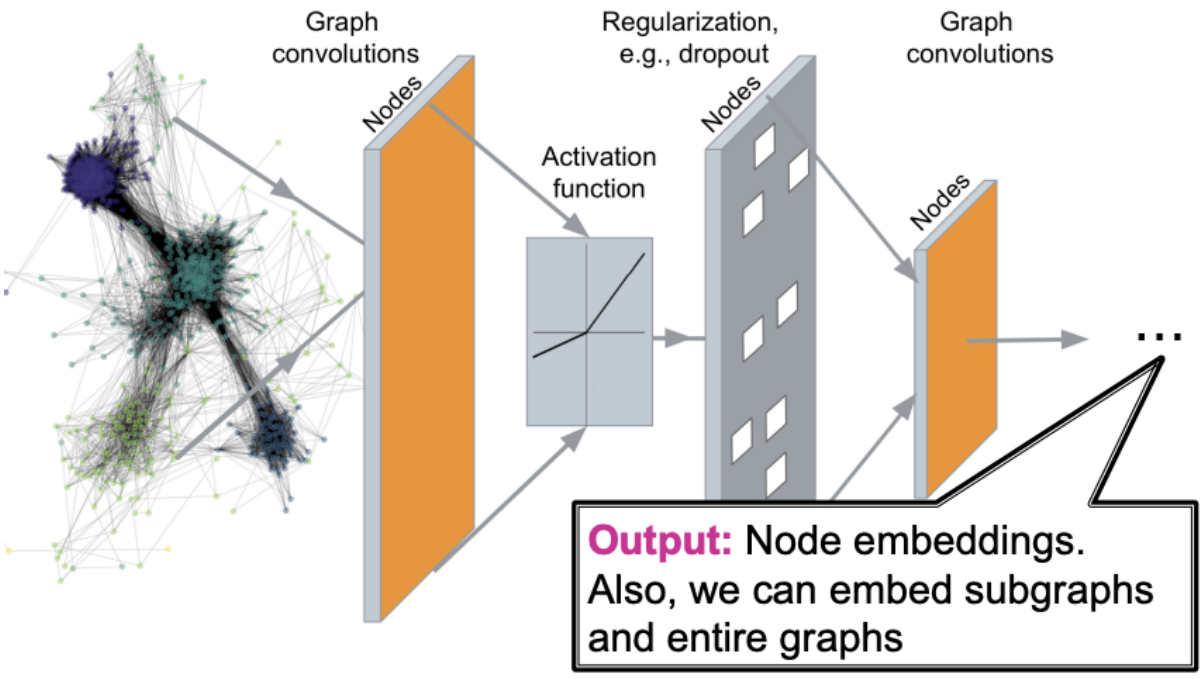
8. GNN Augmentation and Training

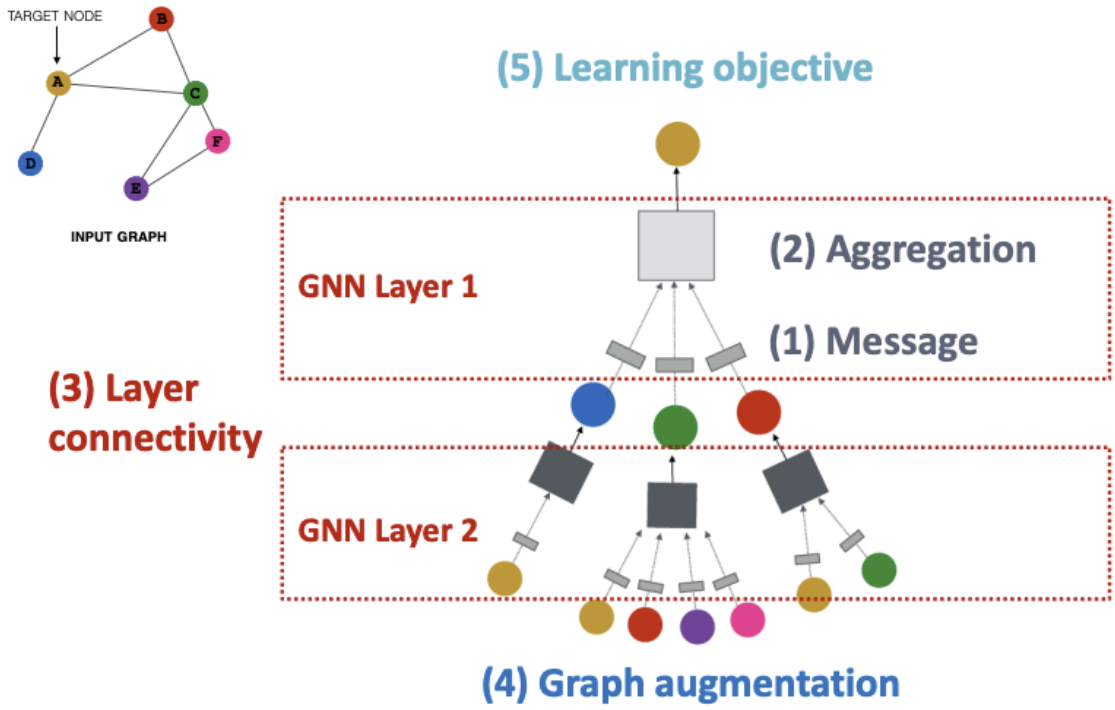
Contents

1. Recap

Recap : Deep Graph Encoders

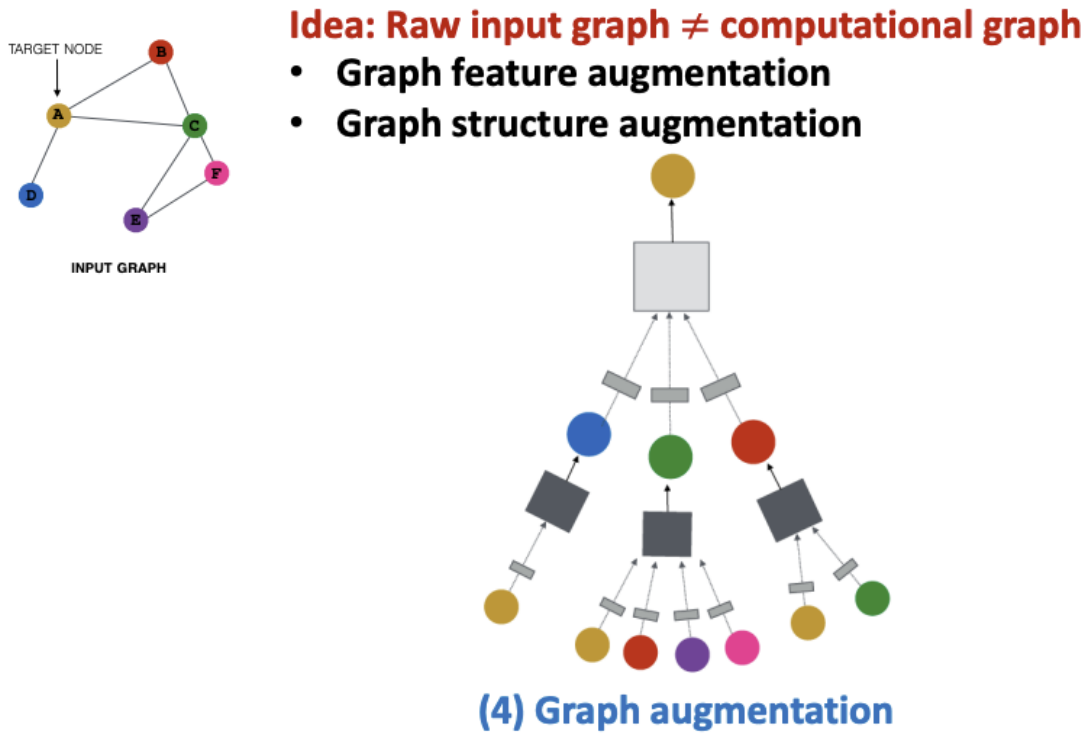


Recap : A General GNN Framework



---

2. Graph Augmentation for GNNs



Why Augment Graphs

Our assumption so far has been

- **Raw input graph = Computational graph**

Reasons for breaking this assumption

- Features:
  - The input graph **lacks features**
- Graph structure:
  - The graph is **too sparse** → inefficient message passing
  - The graph is **too dense** → message passing is too costly
  - The graph is **too large** → cannot fit the computational graph into a GPU
- It's **unlikely that the input graph happens to be the optimal computation graph** for embeddings

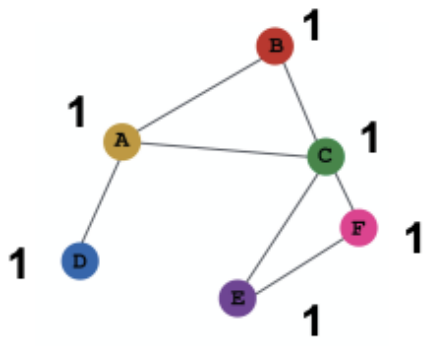
Graph Augmentation Approaches

- **Graph Feature augmentation**
  - The input graph **lacks features** → **feature augmentation**
- **Graph Structure augmentation**
  - The graph is **too sparse** → **Add virtual nodes/edges**
  - The graph is t**oo dense** → **Sample neighbors when doing message passing**
  - The graph is **too large** → **Sample subgraphs to compute embeddings**
    - Will cover later in lecture : Scaling up GNNs

Feature Augmentation on Graphs

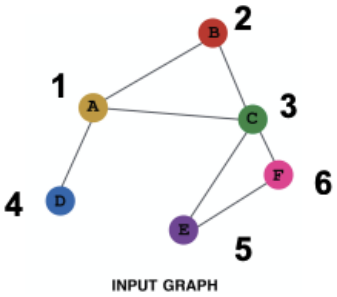
Why do we need feature augmentation?

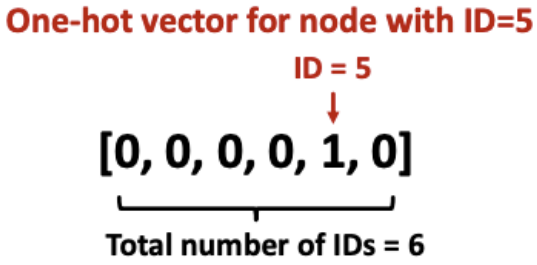
- **(1) Input graph does not have node features**
  - This is common when we only have the adjacency matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**



INPUT GRAPH

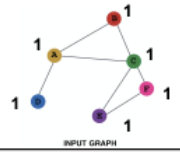
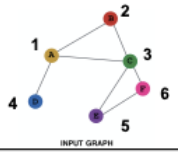
- **b) Assign unique IDs to nodes**
  - These IDs are converted into **one-hot vectors**





- 각 노드에 **상수를 부여**하거나 **one-hot 형태의 ID**를 부여할 수 있다.
- Constant node feature은 1차원이기 때문에 연산량이 적지만 표현력 또한 그만큼 제한적이다. 또한, 그래프의 크기와 무관하게 적용 가능하며 새로운 노드도 쉽게 일반화할 수 있다.
- One-hot node feature은 연산량은 크지만 표현력이 뛰어나며 작은 그래프에만 적용가능하고 새로운 노드는 임베딩할 수 없다.

Feature augmentation: constant vs. one-hot

	Constant node feature	One-hot node feature
		
Expressive power	**Medium**. All the nodes are identical, but GNN can still learn from the graph structure	**High**. Each node has a unique ID, so node-specific information can be stored
Inductive learning (Generalize to unseen nodes)	**High**. Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN	**Low**. Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn't know how to embed unseen IDs
Computational cost	**Low**. Only 1 dimensional feature	**High**. $O(\|V\|)$ dimensional feature, cannot apply to large graphs
Use cases	**Any graph, inductive settings** (generalize to new nodes)	**Small graph, transductive settings** (no new nodes)

※ Transductive learning이란?

그래프 상의 일부 노드와 에지의 ground truth를 아는 상태에서 나머지 노드와 에지의 값을 추정하는 것이다.

즉, known노드와 에지를 가지고 supervised learning을 해서 연결된 unknown 노드와 에지를 prediction하는 방식을 의미한다.

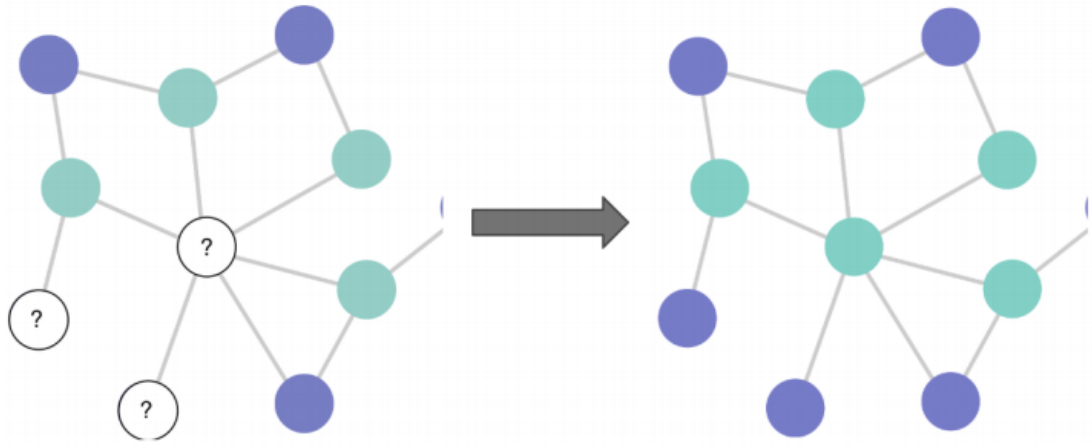
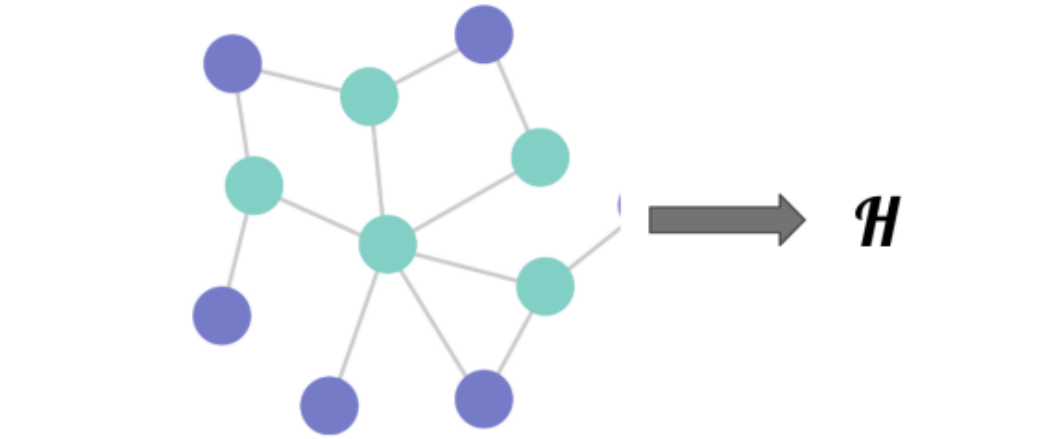


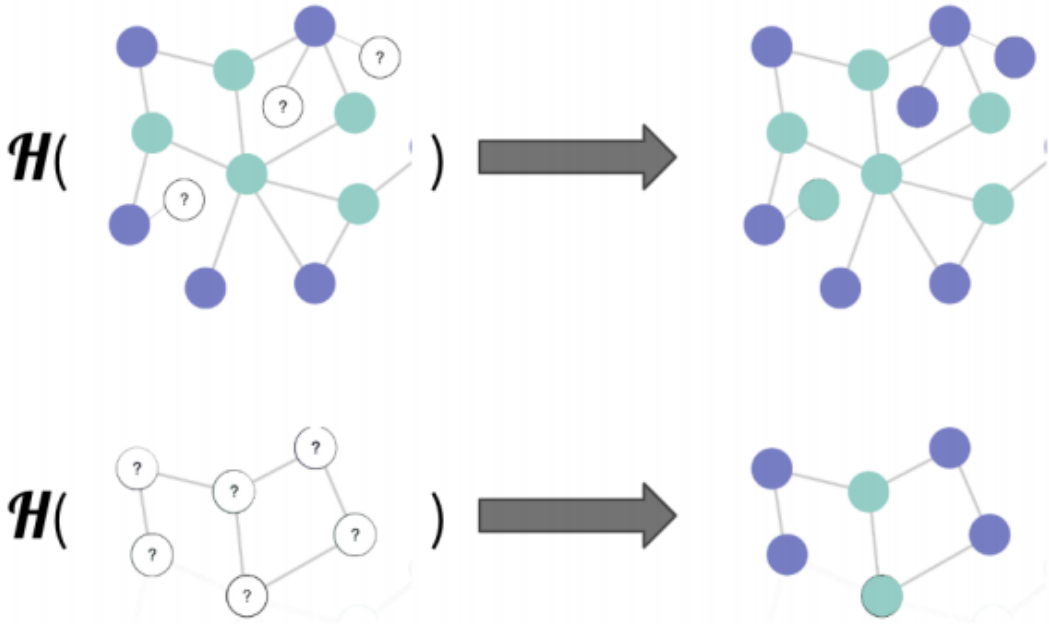
Figure 1. Node classification in transductive setting. At training time, the learning algorithm has access to all the nodes and edges including nodes for which labels are to be predicted.

※ Inductive learning이란?

ground truth를 알고 있는 graph(들)에 대해서 모델을 학습한 후, 전혀 새로운 graph에 대해 노드와 에지의 값을 추정하는 것. 우리가 일반적으로 알고 있는 supervised learning의 형태임



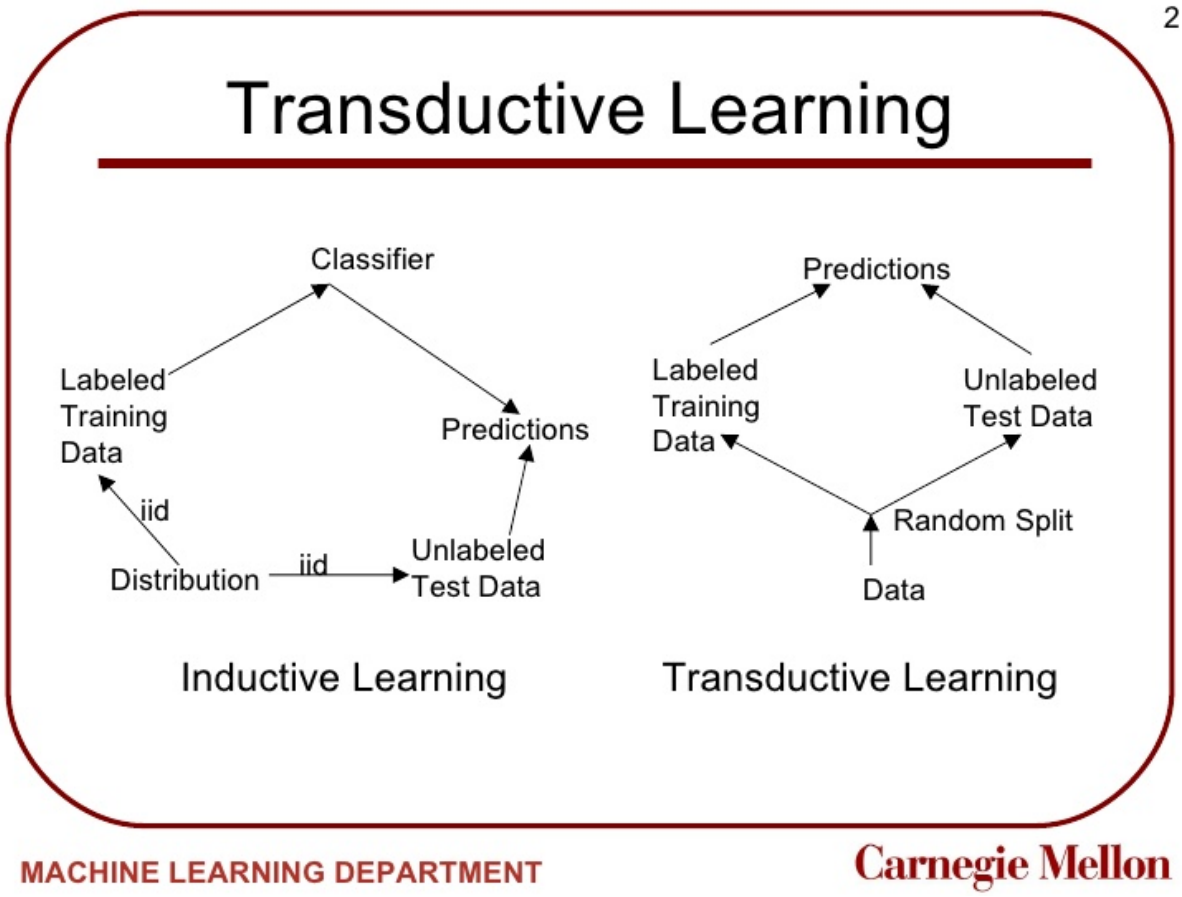
(a) A model $\mathcal{H}$ is learned over some graph



(b) The model is then by applied to new nodes and edges

Figure 2. Node classification in inductive setting. Once learned, the model can be applied to new unseen nodes (outlined in red). There may or may not exist edges between such new nodes and the nodes used for training.

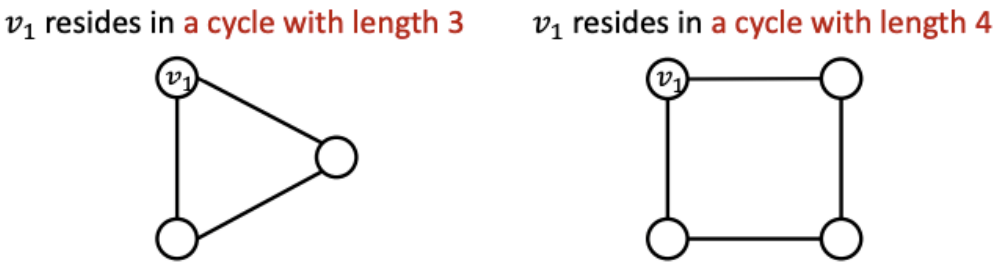
일반적인 머신러닝 관점에서



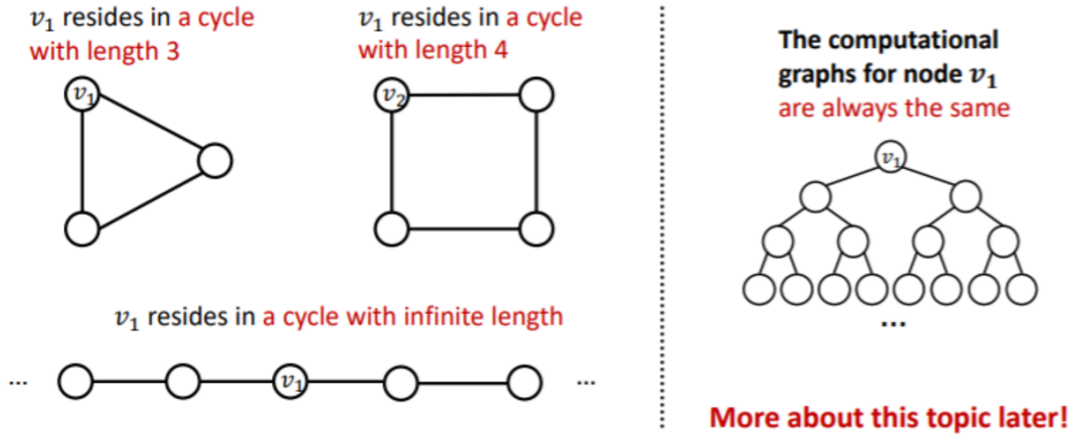
Inductive learning은 일반적인 supervised learning과 동일함. Train-validation-test dataset에 대해서 train과 validation set으로 model을 학습해서 (비록 label 없어서 실제 성능은 알 수 없지만) test set에 대해서 좋은 성능을 나타내는 것을 목표로 하고 있음. 가장 중요한 것은 predictive model을 만든 다는 것.

Transductive learning은 모델을 만들지 않음. 즉, universal한 모델을 만들어서 그 모델의 generalization을 극대화 시켜서 다가올 모르는 문제에 대해서 평균적으로 좋은 성능을 나타내기를 기대하지 않음.

- **(2) Certain structures are hard to learn by GNN**
- **Example : Cycle count feature:**
  - Can GNN learn **the length of a cycle that v_1 resides in?**
  - **Unfortunately, no**

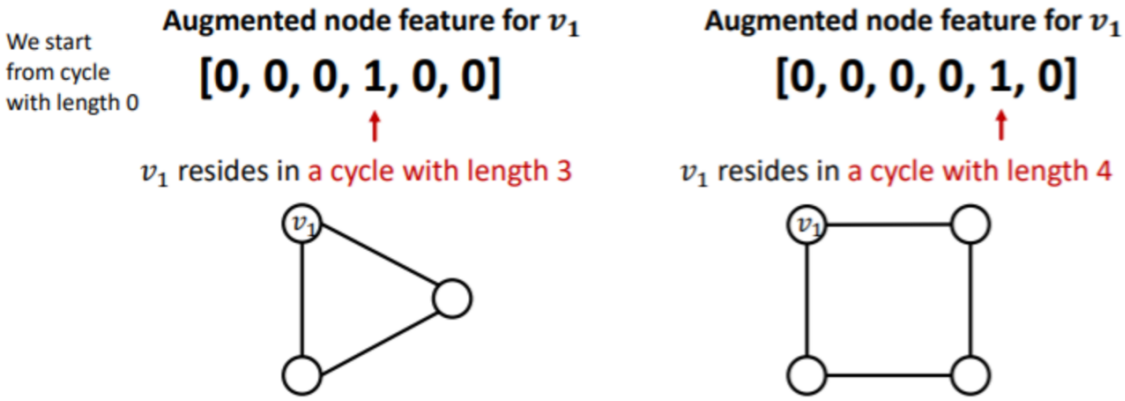


- v_1 cannot differentiate which graph it resides in
  - Because all the nodes in the graph **have degree of 2**
  - The computational graphs will be **the same binary tree**



Circle 구조의 그래프의 경우 노드의 갯수가 달라도 모든 노드가 degree=2로 이루어져 있어 계산 그래프가 binary tree 형태가 되기 때문에 **두 그래프가 구분이 되지 않는다.**

- **Solution:**
  - We can use **cycle count as augmented node features**

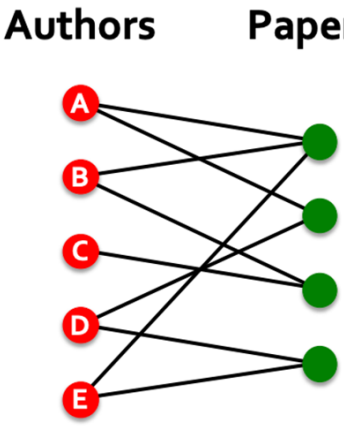


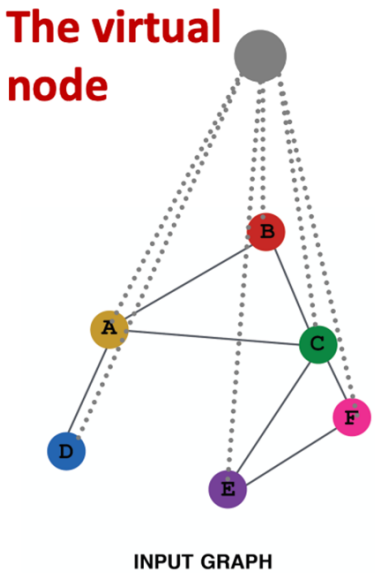
- Other commonly used augmented features:
  - Node degree
  - Clustering coefficient
  - PageRank
  - Centrality
  - …
- **Any feature we have introduced in Lecture 2 can be used!**

Add Virtual Nodes / Edges

- **Motivation : Augment sparse graphs**
- **(1) Add virtual edges**
  - **Common approach** : Connect 2-hop neighbors via virtual edges
  - **Intuition** : Instead of using adj.matrix A for GNN computation, use $A + A^2$
- **그래프가 sparse**한 경우 **가상의 엣지 혹은 노드를 추가**해줄 수 있다.
- 인접행렬의 n승은 n-hop으로 갈 수 있는 노드를 나타내므로 GNN의 입력으로 $A + A^2$을 이용하면 **가상의 2-hop edge**를 추가하는 것이 된다.
- **Use cases : Bipartite graphs**
  - Bipartite graph에서 2-hop은 같은 집단 내 연결을 의미하므로 효과적으로 활용할 수 있다(ex.authors의 2-hop은 공동저자를 의미)
- **(2) Add virtual nodes**

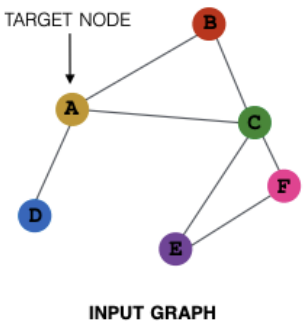
- The virtual node will connect to all the nodes in the graph
  - Suppose in a sparse graph, two nodes have shortest path distance of 10
  - After adding the virtual node, **all the nodes will have a distance of two**
    - Node A - Virtual node - Node B
- **Benefits** : Greatly **imporves message passing in sparse graphs**

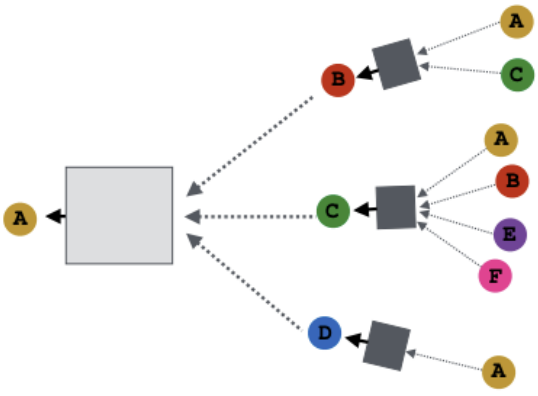




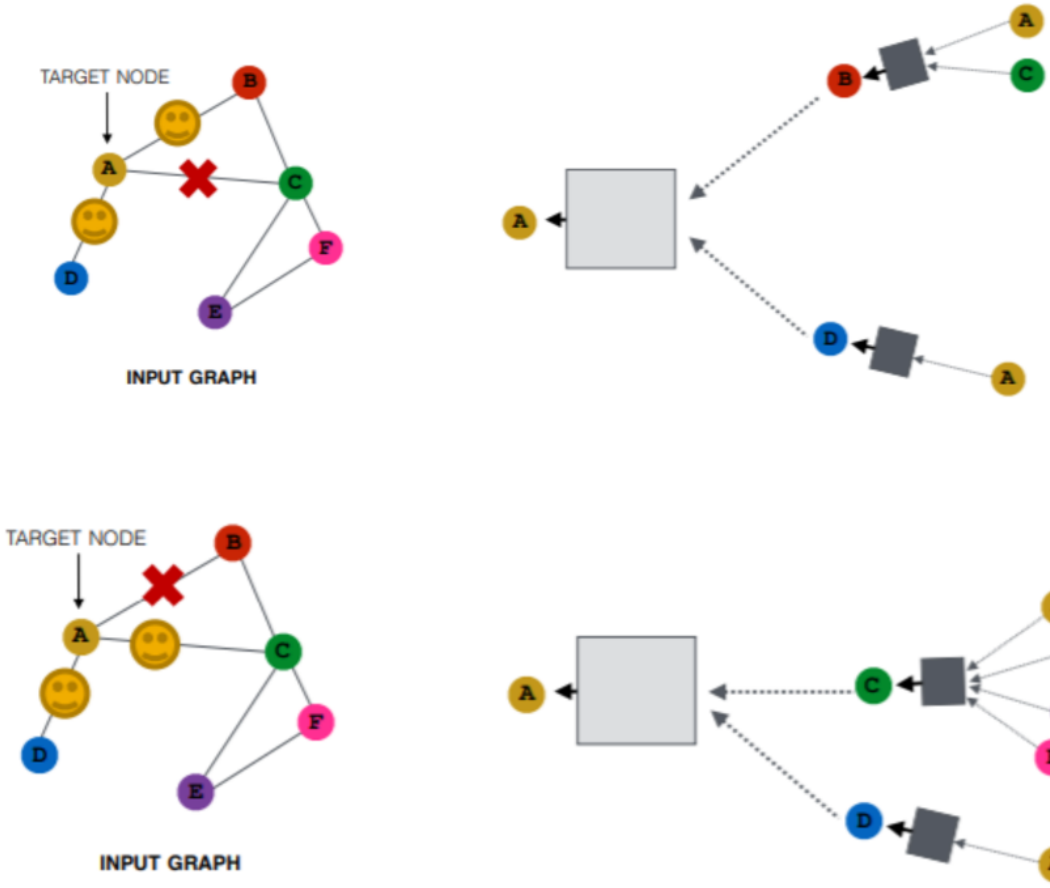
Node Neighborhood Sampling

- **Previously:**
  - All the nodes are used for message passing





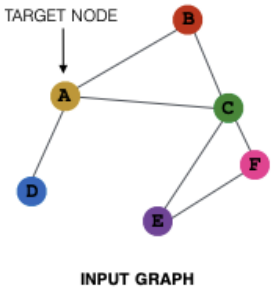
- **New idea** : (Randomly)sample a node's neighborhood for message passing



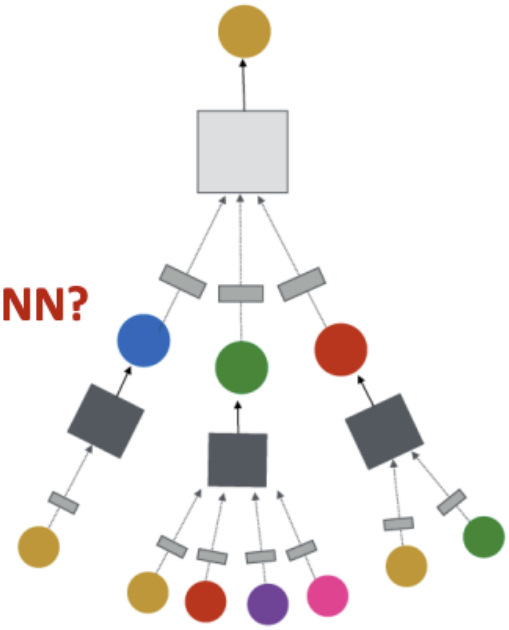
- **그래프가 너무 dense**하여 연산량이 많을 경우 **샘플링을 통해 일부 이웃노드만 사용**할 수 있다.
- 매 에폭, 스텝, 레이어마다 샘플링을 고정하지 않고 다르게 반복하여 **서로 다른 계산 그래프들로부터 노드 임베딩을 형성**할 수 있다.
- **In expectation, we get embeddings similar to the case where all the neighbors are used**
  - **Benefits** : Greatly reduces computational cost
    - **Allows for scaling to large graphs(more about this later)**
  - And in practice it works great!

---

3. Prediction with GNNs



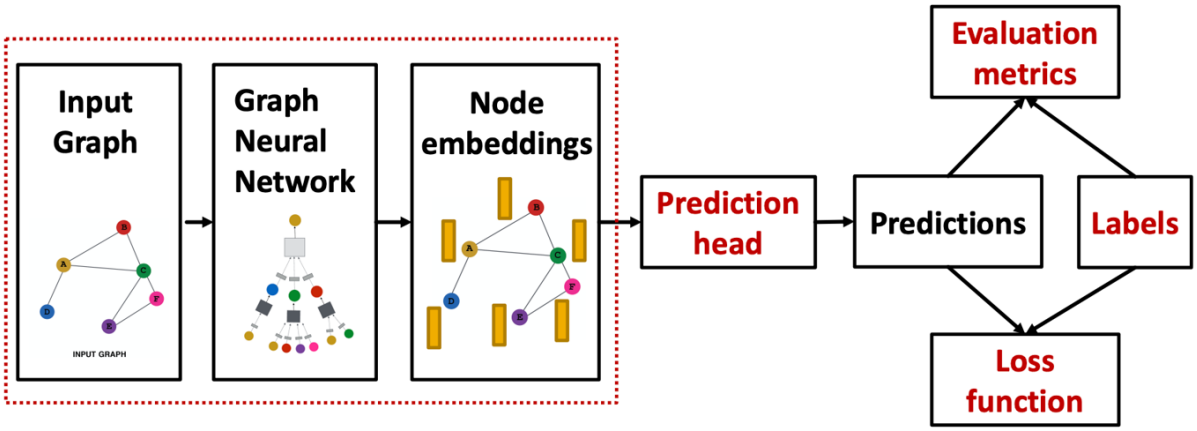
(5) Learning objective



Next: How do we train a GNN?

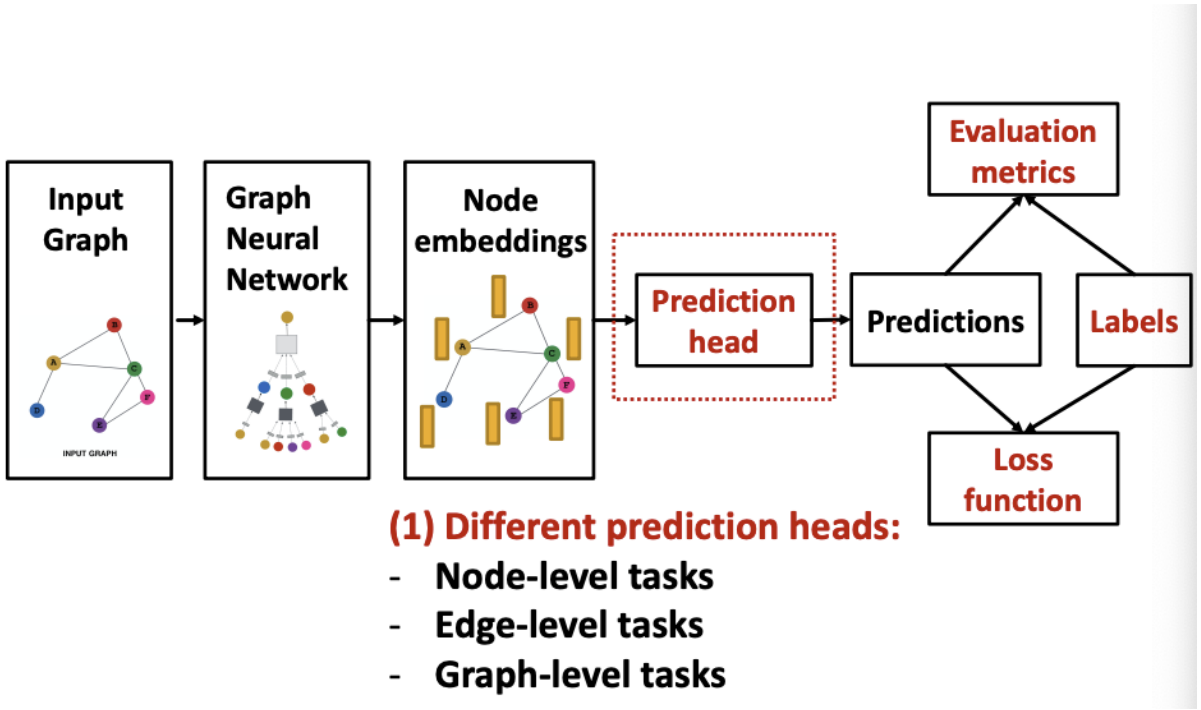
GNN Training Pipeline

So far what we have covered



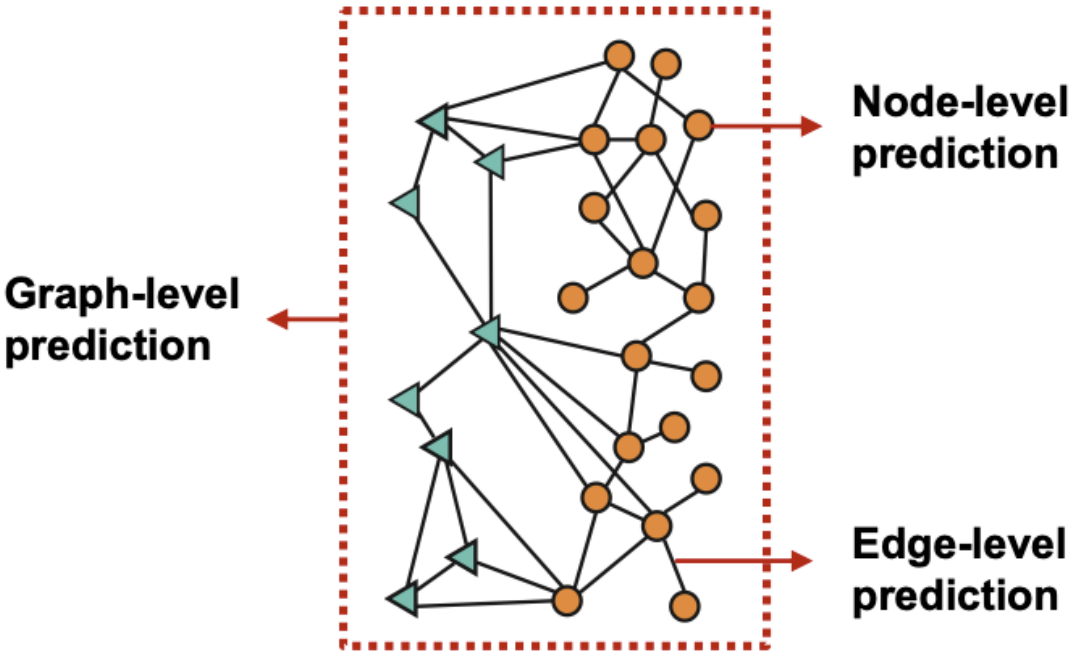
Output of a GNN: set of node embeddings

$$\{\mathbf{h}_v^{(L)}, \forall v \in G\}$$



GNN Prediction Heads

- Idea : Different task levels require different prediction heads

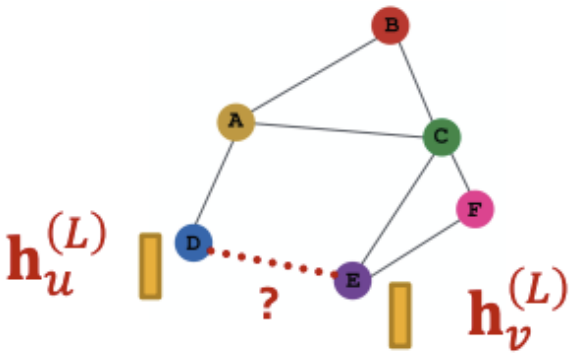


Prediction Heads : Node-level

- **Node-level prediction** : We can directly make prediction using node embeddings!
- After GNN computation, we have **d-dim node embeddings** : $\{h_v^L \in R^d, \forall v \in G\}$
- Suppose we want to make ***k-way prediction***
  - Classification : classify among k categories
  - Regression : regress on k targets
- $\widehat{y_v} = Head_{node}(h_v^{(L)}) = W^{(H)} h_v^{(L)}$
  - $W^{(H)} \in R^{k*d}$: We **map node embeddings from** $h_v^{(L)} \in R^d$ to $\widehat{y_v} \in R^k$ so that we can compute the loss

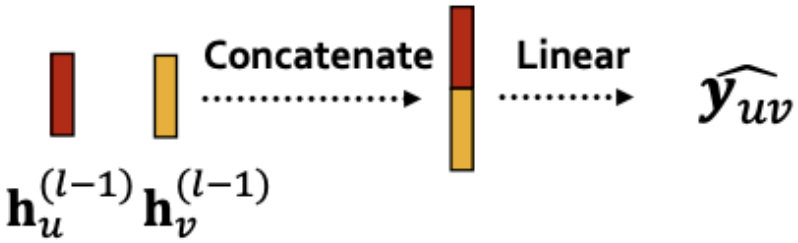
Prediction Heads : Edge-level

- **Edge-level prediction** : Make prediction using pairs of node embeddings
- Suppose we want to make ***k-way prediction***
- $\widehat{y_v} = Head_{edge}(h_u^{(L)}, h_v^{(L)})$



- What are the options for $Head_{edge}(h_u^{(L)}, h_v^{(L)})$ **?**

- **Options for** $Head_{edge}(h_u^{(L)}, h_v^{(L)})$
- **(1) Concatenation + Linear**
  - We have seen this in graph attention



- $\widehat{y_{uv}}$ = **Linear(Concat**$(h_u^{(L)}, h_v^{(L)})$**)**
- Here Linear() will map **2d-dimensional** embeddings(since we concatenated embeddings) to **k-dim** embeddings (k-way prediction)

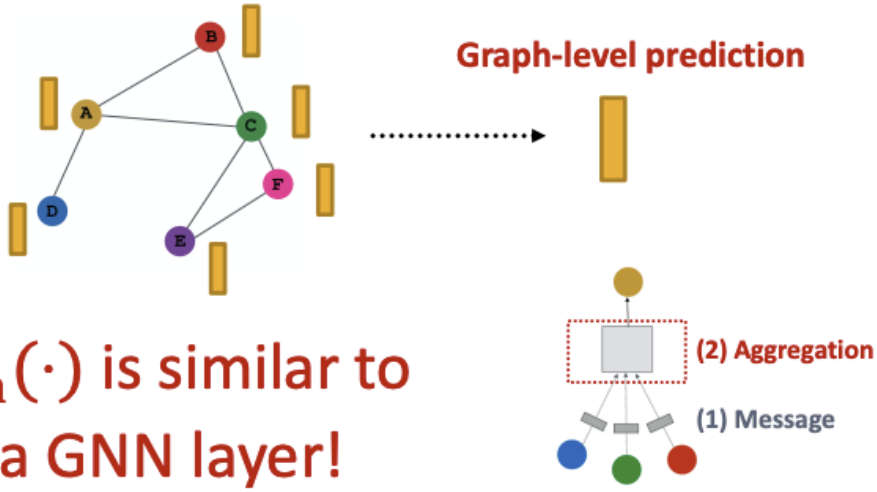
- **(2) Dot product**
  - $\widehat{y_{uv}} = (h_u^{(L)})^T h_v^{(L)}$
  - **This approach only applies to 1-way prediction** (e.g., link prediction : predict the existence of an edge)
  - **Applying to k-way prediction:**
    - Similar to **multi-head attention** :$W^{(1)}, \ldots, W^{(k)}$ trainable

$$\hat{\mathbf{y}}_{uv}^{(1)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(1)} \mathbf{h}_v^{(L)}$$
$$\ldots$$
$$\hat{\mathbf{y}}_{uv}^{(k)} = (\mathbf{h}_u^{(L)})^T \mathbf{W}^{(k)} \mathbf{h}_v^{(L)}$$
$$\hat{\mathbf{y}}_{uv} = \text{Concat}(\hat{\mathbf{y}}_{uv}^{(1)}, \ldots, \hat{\mathbf{y}}_{uv}^{(k)}) \in \mathbb{R}^k$$

Prediction Heads : Graph-level

- **Graph-level prediction** : Make prediction using all the node embeddings in our graph
- Suppose we want to make **k-way prediction**

$$\hat{\mathbf{y}}_G = \text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$



$\text{Head}_{\text{graph}}(\cdot)$ is similar to $\text{AGG}(\cdot)$ in a GNN layer!

- Options for $\text{Head}_{\text{graph}}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$
- **(1) Global mean pooling**

$$\hat{\mathbf{y}}_G = \text{Mean}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(2) Global max pooling**

$$\hat{\mathbf{y}}_G = \text{Max}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- **(3) Global sum pooling**

$$\hat{\mathbf{y}}_G = \text{Sum}(\{\mathbf{h}_v^{(L)} \in \mathbb{R}^d, \forall v \in G\})$$

- These options work great for small graphs
- **Can we do better for large graphs?**

Issue of Global Pooling

- **Issue** : Global pooling over a (large)graph will lose information
- Toy example : we use 1-dim node embeddings
  - Node embeddings for G_1 : {-1, -2, 0, 1, 2}

- Node embeddings for G_2 : {-10, -20, 0, 10, 20}
- **Clearly G_1 and G_2 have very different node embeddings**

  → Their structures should be different

- **If we do global sum pooling :**
  - prediction for G_1 : $\hat{y}_G = 0$
  - prediction for G_2 : $\hat{y}_G = 0$

  → We cannot differentiate G_1 and G_2 !

Hierarchical Global Pooling

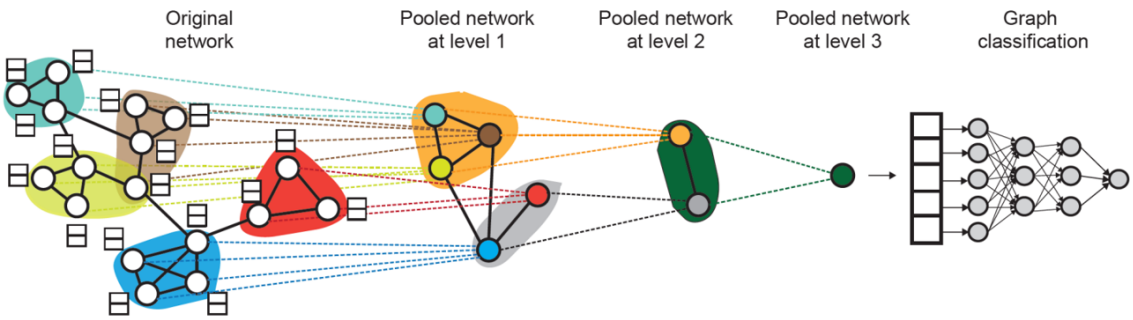
- **A solution:** Let's aggregate all the node embeddings **hierarchically**
  - **Toy example:** We will aggregate via $\mathrm{ReLU}(\mathrm{Sum}(\cdot))$
    - We first separately aggregate the first 2 nodes and last 3 nodes
    - Then we aggregate again to make the final prediction
  - G_1 node embeddings: $\{-1, -2, 0, 1, 2\}$
    - **Round 1:** $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-1, -2\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,1, 2\})) = 3$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{3}$
  - G_2 node embeddings: $\{-10, -20, 0, 10, 20\}$
    - **Round 1:** $\hat{y}_a = \mathrm{ReLU}(\mathrm{Sum}(\{-10, -20\})) = 0$, $\hat{y}_b = \mathrm{ReLU}(\mathrm{Sum}(\{0,10, 20\})) = 30$
    - **Round 2:** $\hat{y}_G = \mathrm{ReLU}(\mathrm{Sum}(\{y_a, y_b\})) = \mathbf{30}$

Now we can differentiate G_1 and G_2 !

https://arxiv.org/pdf/1806.08804.pdf

Hierarchical Pooling In Practice

- **DiffPool idea:**
  - **Hierarchically pool node embeddings**



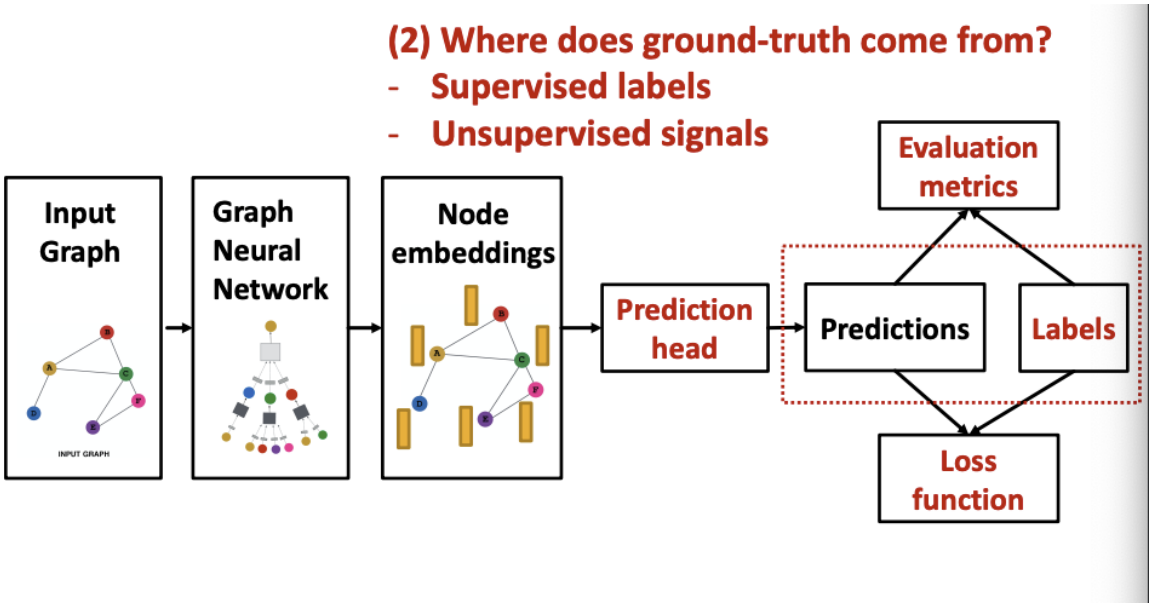
- **Leverage 2 independent GNNs at each level**
  - **GNN A** : Compute node embeddings
  - **GNN B** : Compute the cluster that a node belongs to
- **GNNs A and B at each level can be executed in parallel**

- **For each Pooling layer**
  - Use clustering assignments from **GNN B** to aggregate node embeddings generated by **GNN A**
  - Create a **single new node** for each cluster, maintaining edges between clusters to generated a new **pooled** network
- **Jointly train GNN A and GNN B**

---

4. Training Graph Neural Networks

GNN Training Pipeline (2)



Supervised vs Unsupervised

- **Supervised learning on graphs (레이블이 외부 소스로부터 오는 경우)**
  - **Labels come from external sources**
    - E.g., predict drug likeness of a molecular graph
- **Unsupervised learning on graphs (그래프 내의 정보를 이용하는 경우)**
  - **Signals come from graphs themselves**
    - E.g., link prediction : predict if two nodes are connected
- **Sometimes the differences are blurry**
  - We still have “supervision” in unsupervised learning
    - E.g., train a GNN to predict node clustering coefficient
  - An alternative name for “unsupervised” is “**self-supervised**”
- 그래프에서 **supervised learning은 레이블이 외부 소스로부터 오는 경우**(ex.분자구조에 대한 약효 예측 라벨), **unsupervised learning은 그래프 내의 정보를 이용하는 경우**(ex.두 노드의 엣지 존재 유무)를 의미한다.
- **Unsupervised learning은 경계가 모호해져 self-supervised learning와 동일시되기도 하며** 노드 정보, 엣지 유무 등 그래프 내의 있는 요소 중 지도 학습으로 사용될 수 있는 요소를 찾는 것이 핵심이 된다.

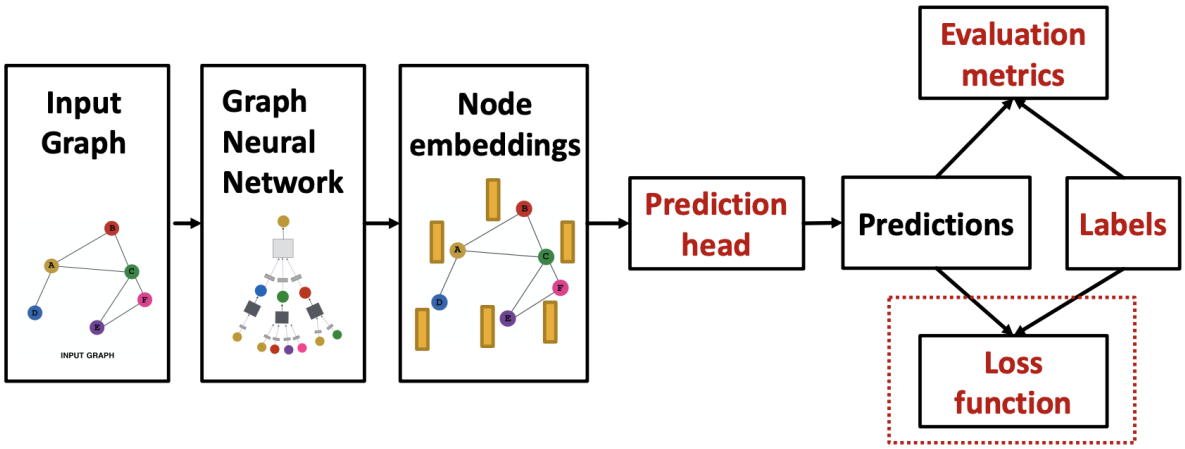
Supervised Labels on Graphs

- **Supervised labels come from the specific use cases.** For example:
  - **Node labels y_v : in a citation network, which subject area does a node belong to**
  - **Edge labels y_{uv} : in a transaction network, whether an edge is fraudulent**
  - **Graph labels y_G : among molecular graphs, the drug likeness of graphs**
- **Advice : Reduce your task to node / edge / graph labels, since they are easy to work with**
  - **E.g**., we knew some nodes form a cluster. We can treat the cluster that a node belongs to as a **node label**

Unsupervised Signals on Graphs

- **The problem :** sometimes **we only have a graph, without any external labels**
- **The solution** : **“Self-supervised learning”, we can find supervision signals within the graph.**
  - For example, we can **let GNN predict the following:**
  - **Node-label y_v. Node statistics : such as clustering coefficient, PageRank,…**
  - **Edge-label y_{uv}. Link prediction : hide the edge between two nodes, predict if there should be a link**
  - **Graph-label y_G. Graph statistics : for example, predict if two graphs are isomorphic**
  - **These tasks do not require any external labels!**

GNN Training Pipeline (3)



(3) How do we compute the final loss?

- **Classification loss**
- **Regression loss**

Settings for GNN Training

- **The setting:** We have N data points
  - Each data point can be a node/edge/graph
  - **Node-level**: prediction $\hat{\boldsymbol{y}}_v^{(i)}$, label $\boldsymbol{y}_v^{(i)}$
  - **Edge-level**: prediction $\hat{\boldsymbol{y}}_{uv}^{(i)}$, label $\boldsymbol{y}_{uv}^{(i)}$
  - **Graph-level**: prediction $\hat{\boldsymbol{y}}_G^{(i)}$, label $\boldsymbol{y}_G^{(i)}$
  - We will use prediction $\hat{\boldsymbol{y}}^{(i)}$, label $\boldsymbol{y}^{(i)}$ to refer **predictions at all levels**

Classification or Regression

- **Classification** : labels $y^{(i)}$ with discrete value
  - E.g., Node classification : Which category does a node belong to

- **Regression** : labels $y^{(i)}$ with continuous value
  - E.g., predict the drug likeness of a molecular graph
- GNNs can be applied to both settings
- **Differences :** **Loss func & Evaluation metrics**

Classification Loss

- As discussed in lecture 6, **cross entropy (CE)** is a very common loss function in classification
- K-way prediction for i-th data point:

$$\mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = -\sum_{j=1}^{K} \boldsymbol{y}_j^{(i)} \log\left(\hat{\boldsymbol{y}}_j^{(i)}\right)$$

Label: $\boldsymbol{y}^{(i)}$ — Prediction: $\hat{\boldsymbol{y}}^{(i)}$ — i-th data point — j-th class

where:

E.g. | 0 | 0 | 1 | 0 | 0 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^K$ = one-hot label encoding

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^K$ = prediction after Softmax$(\cdot)$

E.g. | 0.1 | 0.3 | 0.4 | 0.1 | 0.1 |

- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \mathrm{CE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

Regression Loss

- For regression tasks we often use **Mean Squared Error (MSE)** a.k.a. **L2 loss**
- K-way regression for data point (i):

$$\text{MSE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right) = \sum_{j=1}^{K} \left(\boldsymbol{y}_j^{(i)} - \hat{\boldsymbol{y}}_j^{(i)}\right)^2$$

i-th data point

j-th target

where:

E.g. | 1.4 | 2.3 | 1.0 | 0.5 | 0.6 |

$\boldsymbol{y}^{(i)} \in \mathbb{R}^k$ = Real valued vector of targets

$\hat{\boldsymbol{y}}^{(i)} \in \mathbb{R}^k$ = Real valued vector of predictions

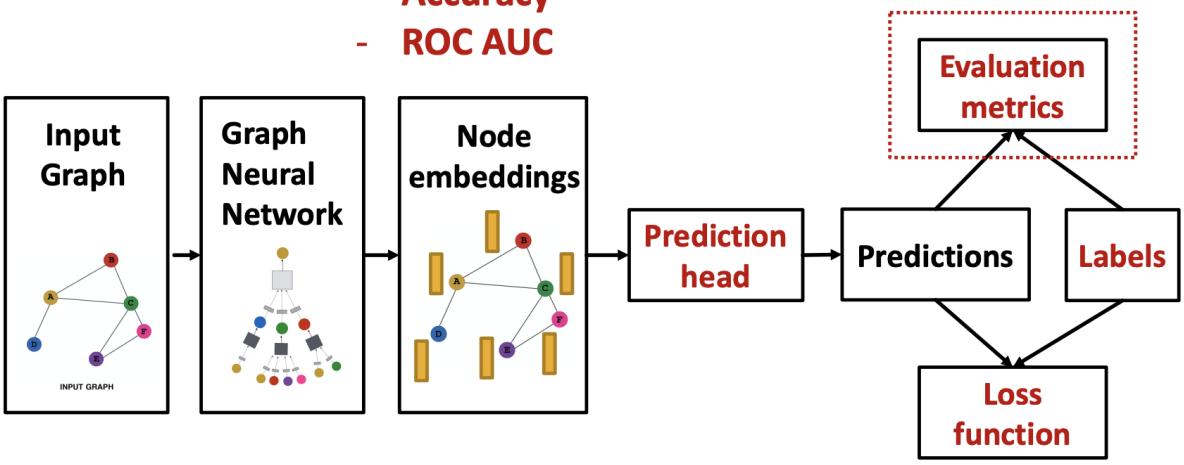
E.g. | 0.9 | 2.8 | 2.0 | 0.3 | 0.8 |

- Total loss over all N training examples

$$\text{Loss} = \sum_{i=1}^{N} \text{MSE}\left(\boldsymbol{y}^{(i)}, \hat{\boldsymbol{y}}^{(i)}\right)$$

GNN Training Pipeline (4)

(4) How do we measure the success of a GNN?
- **Accuracy**
- **ROC AUC**



Evaluation Metrics : Regression

- We use standard evaluation metrics for GNN
  - In practice we will use sklearn for implementation

- Suppose we make predictions for N data points

- Evaluation regression tasks on graphs:

  - **Root mean square error (RMSE)**

$$\sqrt{\sum_{i=1}^{N} \frac{(\boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)})^2}{N}}$$

  - **Mean absolute error (MAE)**

$$\frac{\sum_{i=1}^{N} \left| \boldsymbol{y}^{(i)} - \hat{\boldsymbol{y}}^{(i)} \right|}{N}$$

Evaluation Metrics : Classification

- **Evaluate classification tasks on graphs:**
- **(1) Multi-class classification**
  - **We simply report the accuracy**
  - $\frac{1[argmax(\hat{y}^{(i)})=y^{(i)}]}{N}$
- **(2) Binary classification**
  - Metrics sensitive to classification threshold
    - **Accuracy**
    - **Precision / Recall**
    - If the range of prediction is [0, 1], we will use 0.5 as threshold
  - Metric Agnostic to classification threshold
    - **ROC AUC**

Metrics for Binary Classification

- **Accuracy**:

$$\frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{|Dataset|}$$

- **Precision (P)**:

$$\frac{TP}{TP + FP}$$

- **Recall (R)**:

$$\frac{TP}{TP + FN}$$

- **F1-Score**:

$$\frac{2P * R}{P + R}$$

Confusion matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Evaluation Metrics

- **ROC Curve** : Captures the tradeoff in TPR and FPR **as the classification threshold is varied for a binary classifier.**

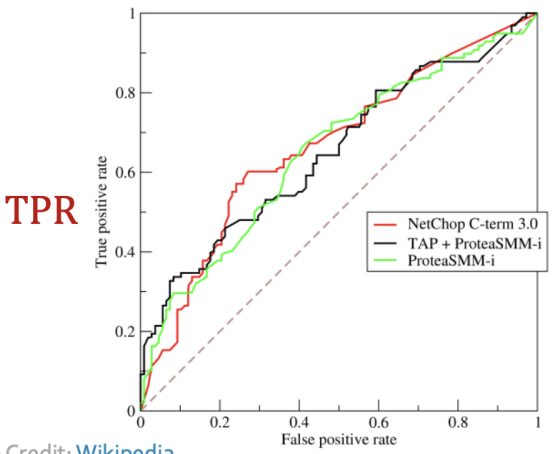


Image Credit: Wikipedia

$$TPR = Recall = \frac{TP}{TP + FN}$$

$$FPR = \frac{FP}{FP + TN}$$

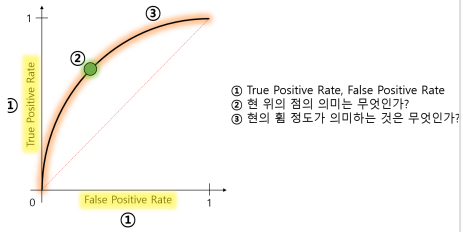
Note: the dashed line represents **performance of a random classifier**

- **ROC AUC** : **Area under the ROC Curve.**
- **Intuition** : The probability that a classifier will rank a randomly chosen positive higher than a randomly chosen negative one

ROC curve

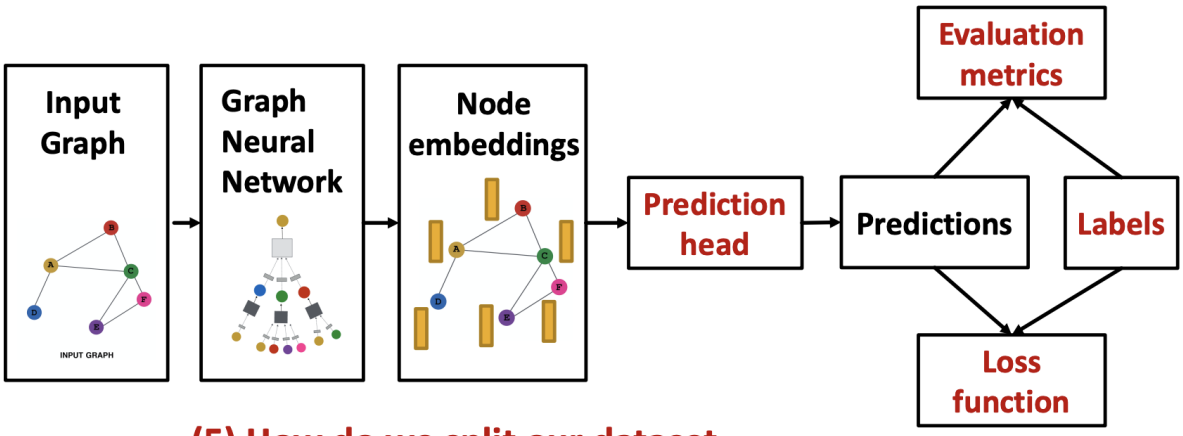
x축은 이진 분류기에 의해 결정된 score가 표시된 것으로 생각할 수 있음. 빨간색과 파란색의 정규분포로 표현한 종모양의 분포들은 데이터 샘플들의 실제 클래스를 나타냄 왼쪽 패널에 있는 흰색 바는 마우스 드래

https://angeloyeo.github.io/2020/08/05/ROC.html

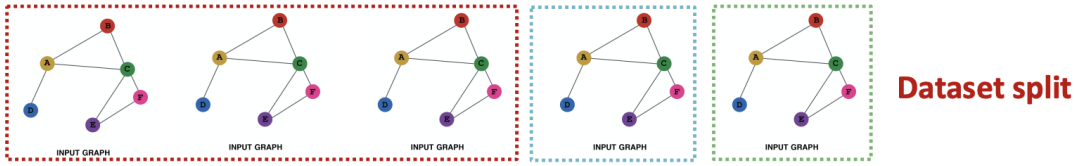


5. Setting-up GNN Prediction Tasks

GNN Training Pipeline (5)



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



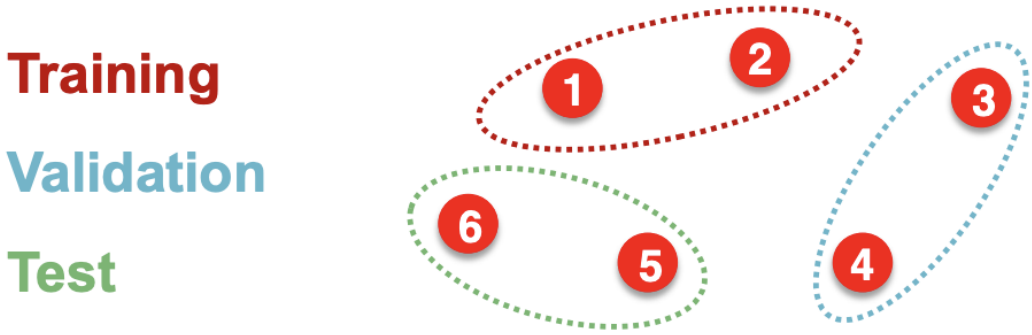
Dataset Split : Fixed / Random Split

- **Fixed split** : We will split our dataset once
  - **Training set** : used for optimizing GNN parameters
  - **Validation set** : develop model/hyperparameters
  - **Test set** : held out until we report final performance
- **A concern** : sometimes we cannot guarantee that the test set will really be held out

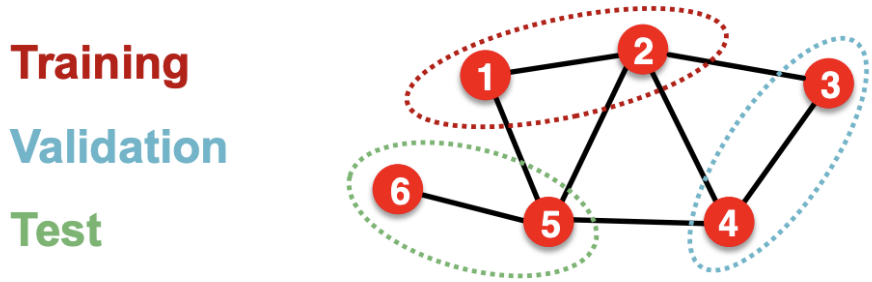
- **Random Split** : we will **randomly split** our dataset into training / validation / test
  - We report **average performance over different random seeds**

Why Splitting Graphs is Special

- **Suppose we want to split an image dataset**
  - **Image classification** **: Each data point is an image**
  - Here **data points are independent**
    - **Image 5 will not affect our prediction on image 1**

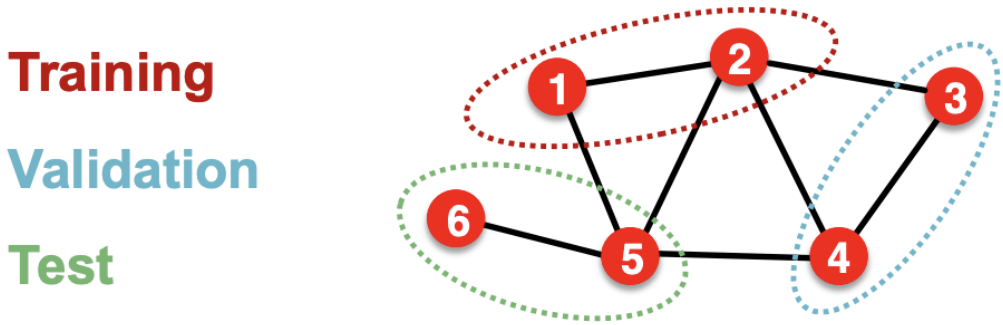


- **Splitting a graph dataset is different!**
  - **Node classification** **: Each data point is a node**
  - Here **data points are NOT independent**
    - **Node 5 will affect our prediction on node 1,** because it will participate in message passing → affect node 1's embedding



- **What are our options?**

- **Solution 1 (Transductive setting) : The input graph can be observed in all the dataset splits(training, validation and test set).**
- We will only split the (node) labels
  - **At training time**, we compute embeddings **using the entire graph**, and **train using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the entire graph**, and **evaluate on node 3&4's labels**

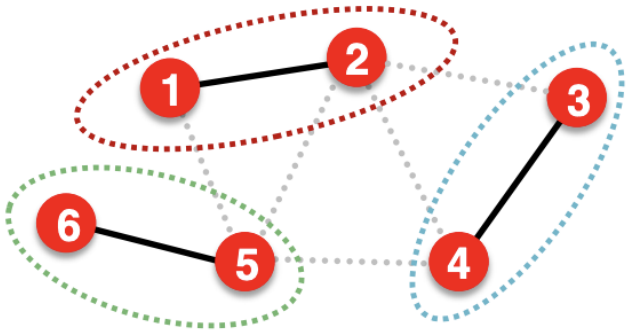


- **Solution 2 (Inductive setting) : We break the edges between splits to get multiple graphs**
  - **Now we have 3 graphs that are independent. Node 5 will not affect our prediction on node 1 anymore**
  - **At training time**, we compute embeddings **using the graph over node 1&2**, and train **using node 1&2's labels**
  - **At validation time**, we compute embeddings **using the graph over node 3&4**, and **evaluate on node 3&4's labels**

Training

Validation

Test

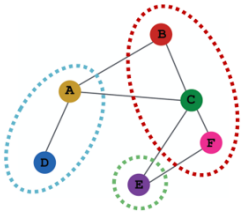


Transductive / Inductive Settings

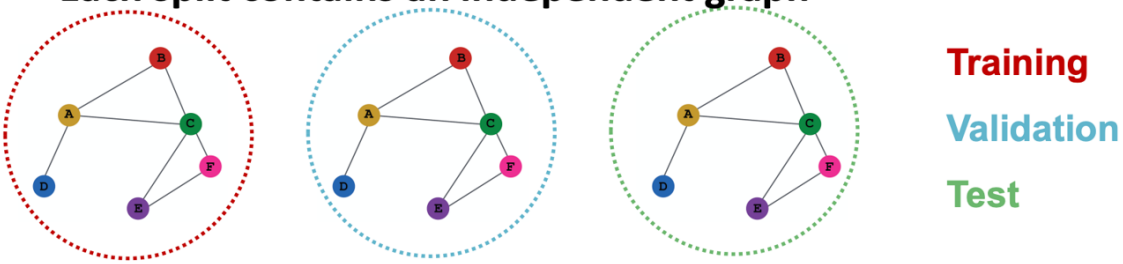
- **Transductive setting** : training / validation / test sets are **on the same graph**
  - The **dataset consists of one graph**
  - **The entire graph can be observed in all dataset splits, we only split the labels**
  - Only applicable to **node / edge** prediction tasks
- **Inductive setting** : training / validation / test sets are **on different graphs**
  - The **dataset consists of multiple graphs**
  - Each split can **only observe the graph(s) within the split.** A successful model should **generalize to unseen graphs**
  - Applicable to **node / edge / graph** task

Example : Node Classification

- **Transductive** node classification
  - **All the splits can observe the entire graph structure**, but can only observe the labels of their respective nodes

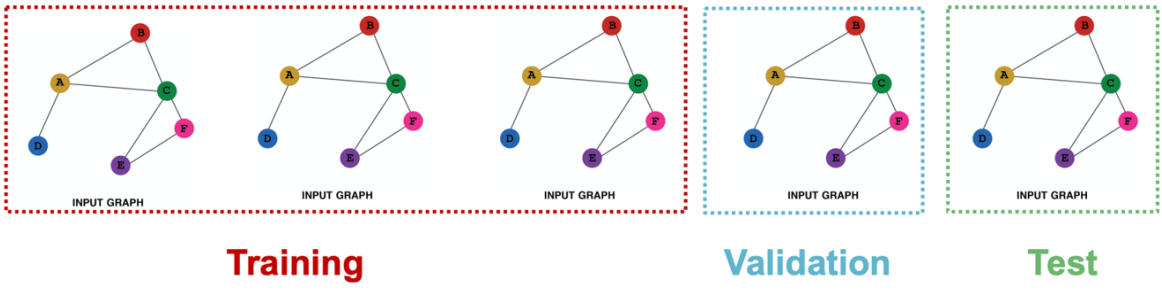


- **Inductive** node classification
  - Suppose we have a dataset of 3 graphs
  - **Each split contains an independent graph**



Example : Graph Classification

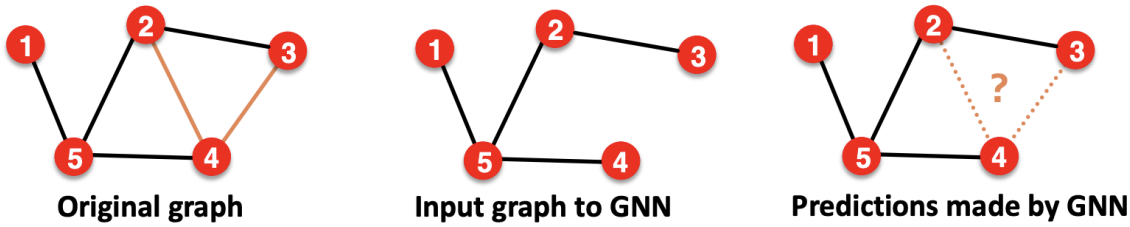
- Only the **inductive setting** is well defined for **graph classification**
  - Because **we have to test on unseen graphs**
  - Suppose we have a dataset of 5 graphs. Each split will contain independent graph(s).



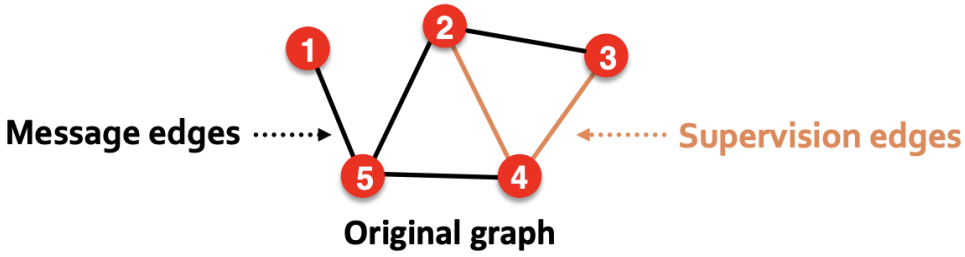
Example : Link Prediction

- **Goal of link prediction : predict missing edges**

- **Setting up link prediction is tricky:**
  - Link prediction is an unsupervised / self-supervised task. We need to **create the labels** and **dataset splits** on our own
  - Concretely, we need to **hide some edges from the GNN and let the GNN predict if the edges exist**

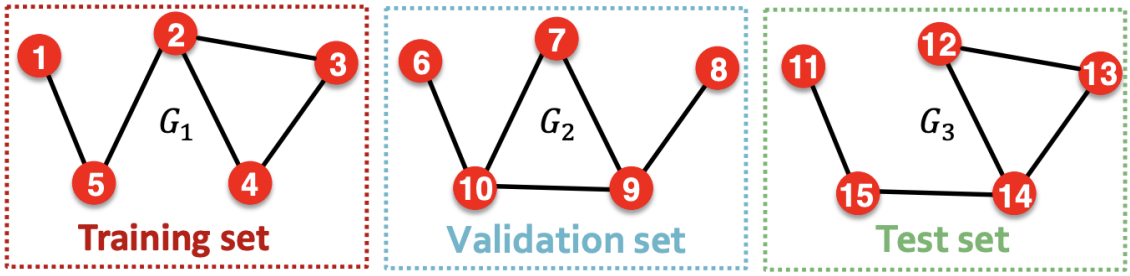


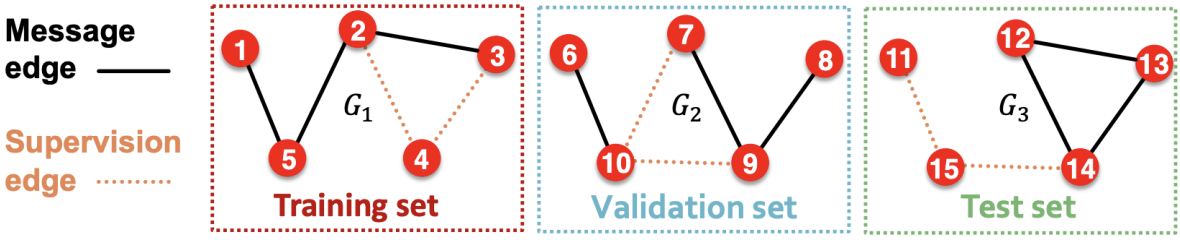
Setting up Link Prediction



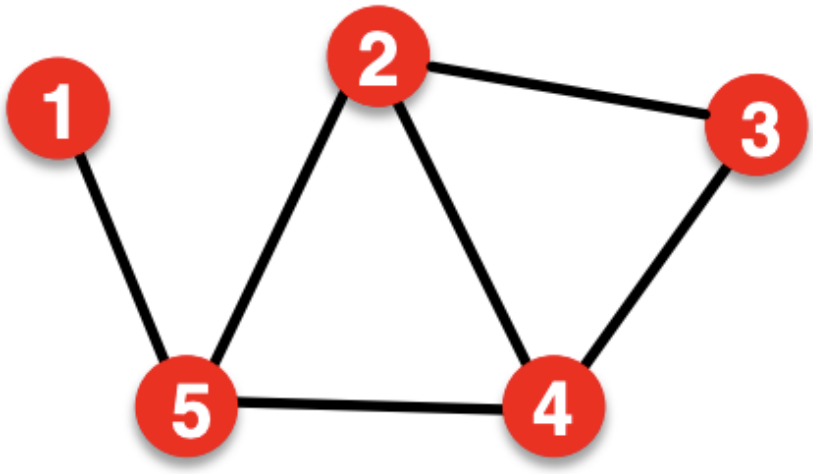
- **For link prediction, we will split edges twice**
- **Step 1 : Assign 2 types of edges in the original graph**
  - **Message edges : Used for GNN message passing**
  - **Supervision edges : Use for computing objectives**
  - **After step 1 :**
    - **Only message edges will remain in the graph**
    - **Supervision edges are used as supervision for edge predictions made by the model, will not be fed into GNN!**
- **Step 2 : Split edges into train / validation / test**
- **Option 1 : Inductive link prediction split**

- **Suppose we have a dataset of 3 graphs. Each inductive split will contain an independent graph**
- In **train** or **val** or **test** set, each graph will have **2** types of edges : message edges + **supervision edges**
  - **Supervision edges** are not the input to GNN

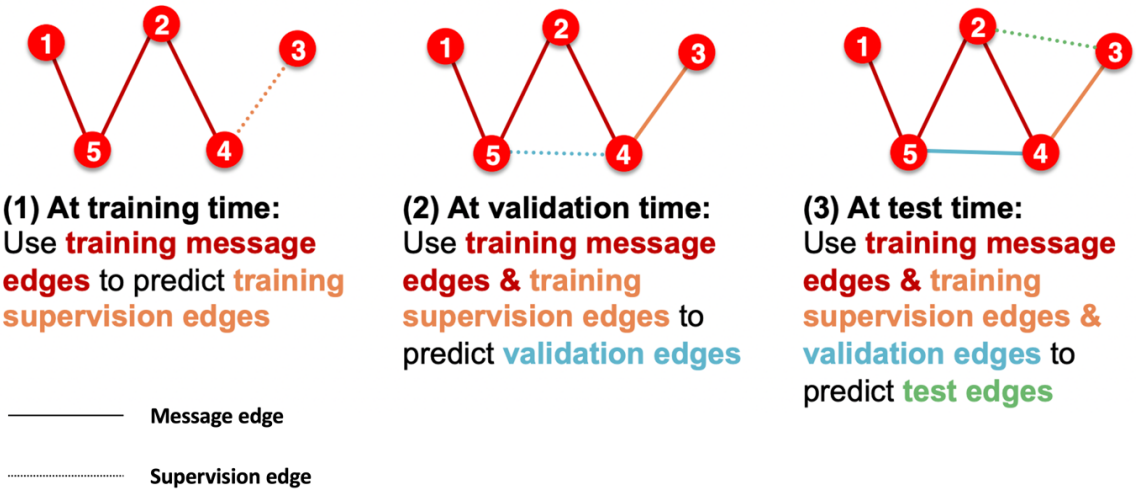




- **Option 2 : Transductive link prediction split:**
  - **This is the default setting when people talk about link prediction**
  - **Suppose we have a dataset of 1 graph**



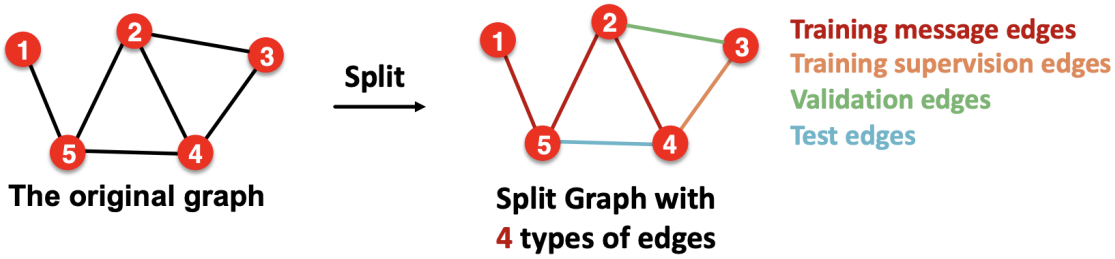
- **By definition of "transductive", the entire graph can be observed in all dataset splits**
  - **But since edges are both part of graph structure and the supervision, we need to hold out validation / test edges**
  - **To train the training set, we further need to hold out supervision edges for the training set**



Why do we use growing number of edges?

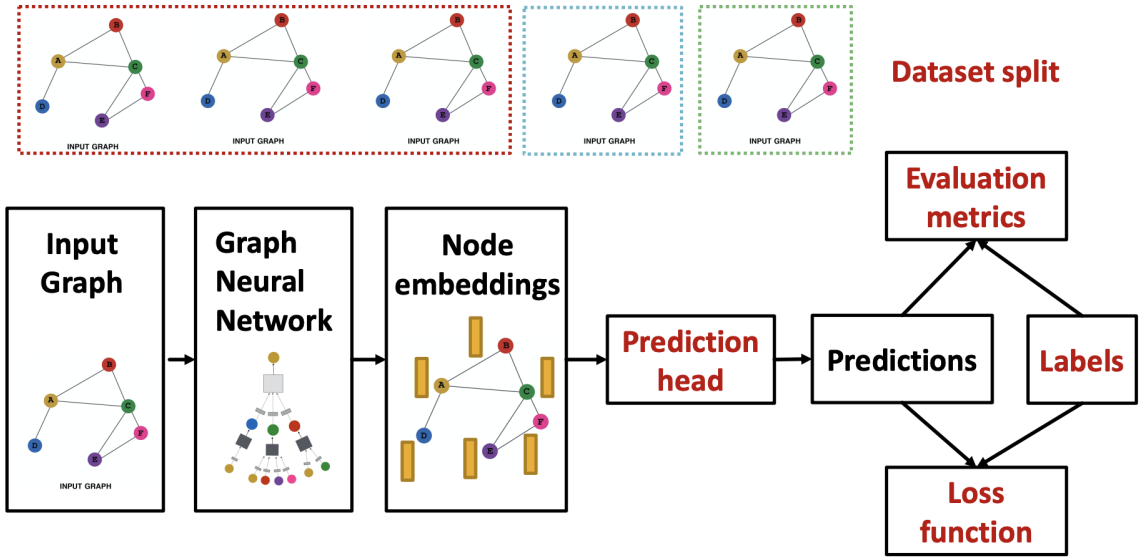
After training, supervision edges are known to GNN.
Therefore, an ideal model should use supervision edges in message passing at validation time.
The same applies to the test time.

- **Summary : Transductive link prediction split :**



- **Note** : Link prediction settings are tricky and cmplex. You may find paaers do link prediction differently. But if you follow our reasoning steps, **this should be thr right way to implement link prediction**
- Luckily, we have full support in DeepSNAP and GraphGym

GNN Training Pipeline



Implementation resources:

DeepSNAP provides core modules for this pipeline

GraphGym further implements the full pipeline to facilitate GNN design

---

Summary of the Lecture

- We introduce a general perspective for GNNs
  - GNN Layer :
    - Transformation + Aggregation
    - Classic GNN layers : GCN, GraphSAGE, GAT
  - Layer connectivity:
    - The over-smoothing problem
    - Solution : skip connections
  - Graph Augmentation :
    - Feature augmentation
    - Structure augmentation
  - Learning Objectives
    - The full training pipeline of a GNN

References

- **Lecture 8.1:** https://www.youtube.com/watch?v=1A6VoEkQnhQ&list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn&index=23
- **Lecture 8.2:**https://www.youtube.com/watch?v=eXIIH8YVxKI&list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn&index=24
- **Lecture 8.3:**https://www.youtube.com/watch?v=ewEW_EMzRuo&list=PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn&index=25
- https://velog.io/@kimkj38/CS224W-Lecture-8.-Applications-of-Graph-Neural-Networks