Stanford CS224W: Setting-up GNN Prediction Tasks
(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*
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 any more
- 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
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 tasks
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
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 the **let the GNN predict if the edges exist**
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!
Setting up Link Prediction

- **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
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.

**Diagram:**

3 graphs: $G_1$, $G_2$, $G_3$

- **Message edge**
  - $G_1$: (1, 2), (2, 3)
  - $G_2$: (6, 7), (7, 8)
  - $G_3$: (12, 13)

- **Supervision edge**
  - $G_1$: (1, 5), (5, 4)
  - $G_2$: (6, 10), (10, 9)
  - $G_3$: (12, 15), (15, 14)
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
Option 2: Transductive link prediction split:

- 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

- Next: we will show the exact settings
### Setting up Link Prediction

- **Option 2: Transductive link prediction split:**

  ![The original graph](image)

  **(1) At training time:**
  Use **training message edges** to predict **training supervision edges**

  **(2) At validation time:**
  Use **training message edges & training supervision edges** to predict **validation edges**

  **(3) At test time:**
  Use **training message edges & training supervision edges & validation edges** to predict **test edges**
Option 2: Transductive link prediction split:

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.

(1) At training time: Use training message edges to predict training supervision edges.

(2) At validation time: Use training message edges & training supervision edges to predict validation edges.

(3) At test time: Use training message edges & validation edges to predict test edges.
Setting up Link Prediction

- Summary: Transductive link prediction split:

  ![Graph](image)

  The original graph

  Split Graph with 4 types of edges

  - Training message edges
  - Training supervision edges
  - Validation edges
  - Test edges

- Note: Link prediction settings are tricky and complex. You may find papers do link prediction differently. But if you follow our reasoning steps, **this should be the right way to implement link prediction**

- Luckily, we have full support in [DeepSNAP](http://deepSNAP.readthedocs.io) and [GraphGym](https://graphgym.readthedocs.io)
GNN Training Pipeline

**Input Graph** → **Graph Neural Network** → **Node embeddings** → **Prediction head** → **Predictions** → **Evaluation metrics** → **Loss function** → **Labels**

**Dataset split**

**Implementation resources:**
- **DeepSNAP** provides core modules for this pipeline
- **GraphGym** further implements the full pipeline to facilitate GNN design
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
Stanford CS224W: How Expressive are Graph Neural Networks?
Output: Node embeddings. We can also embed larger network structures, subgraphs, graphs.
**Idea: Aggregate Neighbors**

- **Key idea:** Generate node embeddings based on local network neighborhoods

![Input Graph](image)

**TARGET NODE**

**INPUT GRAPH**
Idea: Aggregate Neighbors

- **Intuition:** Nodes aggregate information from their neighbors using neural networks
How powerful are GNNs?

- Many GNN models have been proposed (e.g., GCN, GAT, GraphSAGE, design space).

- What is the expressive power (ability to distinguish different graph structures) of these GNN models?

- How to design a maximally expressive GNN model?
Many GNN models have been proposed:
- GCN, GraphSAGE, GAT, Design Space etc.

Different GNN models use different neural networks in the box.
GNN Model Example (1)

- GCN (mean-pool) [Kipf and Welling ICLR 2017]

Element-wise mean pooling + Linear + ReLU non-linearity
GNN Model Example (2)

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]

```
INPUT GRAPH

TARGET NODE

MLP + element-wise max-pooling
```
Note: Node Colors

- We use node same/different **colors** to represent nodes with same/different features.
  - For example, the graph below assumes all the nodes share the same feature.

![Graph Diagram]

- **Key question**: How well can a GNN distinguish different graph structures?
We specifically consider **local neighborhood structures** around each node in a graph.

**Example:** Nodes 1 and 5 have **different** neighborhood structures because they have different node degrees.
We specifically consider local neighborhood structures around each node in a graph.

Example: Nodes 1 and 4 both have the same node degree of 2. However, they still have different neighborhood structures because their neighbors have different node degrees.

Node 1 has neighbors of degrees 2 and 3. Node 4 has neighbors of degrees 1 and 3.
We specifically consider **local neighborhood structures** around each node in a graph.

- **Example:** Nodes 1 and 2 have the **same** neighborhood structure because they are symmetric within the graph.

Node 1 has neighbors of degrees 2 and 3.
Node 2 has neighbors of degrees 2 and 3.
And even if we go a step deeper to 2\(^{nd}\) hop neighbors, both nodes have the same degrees (Node 4 of degree 2)
Key question: Can GNN node embeddings distinguish different node’s local neighborhood structures?
- If so, when? If not, when will a GNN fail?

Next: We need to understand how a GNN captures local neighborhood structures.
- Key concept: Computational graph
In each layer, a GNN aggregates neighboring node embeddings.

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

**Ex:** Node 1’s computational graph (2-layer GNN)
**Ex:** Nodes 1 and 2’s computational graphs.
Ex: Nodes 1 and 2’s computational graphs.
But GNN only sees node features (not IDs):
A GNN will generate the same embedding for nodes 1 and 2 because:

- Computational graphs are the same.
- Node features (colors) are identical.

Note: GNN does not care about node ids, it just aggregates features vectors of different nodes.

GNN won’t be able to distinguish nodes 1 and 2
In general, different local neighborhoods define different computational graphs.
Computational graphs are identical to rooted subtree structures around each node.

Rooted subtree structures
(defined by recursively unfolding neighboring nodes from the root nodes)
GNN’s node embeddings capture rooted subtree structures. Most expressive GNN maps different rooted subtrees into different node embeddings (represented by different colors).
**Recall: Injective Function**

- **Function** $f: X \rightarrow Y$ is **injective** if it maps different elements into different outputs.
- **Intuition:** $f$ retains all the information about input.
Most expressive GNN should map subtrees to the node embeddings *injectively*.
**Key observation**: Subtrees of the same depth can be recursively characterized from the leaf nodes to the root nodes.

Input features are uniform

From leaves to the root

1

\[ (2 \text{ neighbors, } 3 \text{ neighbors}) \]

\[ (1 \text{ neighbor, } 3 \text{ neighbors}) \]

\[ \neq \]

From leaves to the root

1 neighbor

\[ (1 \text{ neighbor, } 3 \text{ neighbors}) \]

\[ (1 \text{ neighbor, } 3 \text{ neighbors}) \]

\[ \neq \]

From leaves to the root

1 neighbor

\[ (1 \text{ neighbor, } 3 \text{ neighbors}) \]

\[ (1 \text{ neighbor, } 3 \text{ neighbors}) \]
If each step of GNN’s aggregation can fully retain the neighboring information, the generated node embeddings can distinguish different rooted subtrees.

Fully retain neighboring information

Input features are uniform

(2 neighbors, 3 neighbors) ≠ (1 neighbor, 3 neighbors)
In other words, most expressive GNN would use an **injective neighbor aggregation** function at each step.

- Maps different neighbors to different embeddings.
Summary so far

To generate a node embedding, GNNs use a computational graph corresponding to a subtree rooted around each node.

- GNN can fully distinguish different subtree structures if every step of its neighbor aggregation is injective.
Stanford CS224W: Designing the Most Powerful Graph Neural Network
Key observation: Expressive power of GNNs can be characterized by that of neighbor aggregation functions they use.

- A more expressive aggregation function leads to a more expressive a GNN.
- Injective aggregation function leads to the most expressive GNN.

Next:
- Theoretically analyze expressive power of aggregation functions.
Observation: Neighbor aggregation can be abstracted as a function over a multi-set (a set with repeating elements).
Next: We analyze aggregation functions of two popular GNN models

- **GCN** (mean-pool) [Kipf & Welling, ICLR 2017]
  - Uses *element-wise* mean pooling over neighboring node features
    \[
    \text{Mean}(\{x_u\}_{u \in N(v)})
    \]

- **GraphSAGE** (max-pool) [Hamilton et al. NeurIPS 2017]
  - Uses *element-wise* max pooling over neighboring node features
    \[
    \text{Max}(\{x_u\}_{u \in N(v)})
    \]
Neighbor Aggregation: Case Study

- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - Take element-wise mean, followed by linear function and ReLU activation, i.e., $\max(0, x)$.

- **Theorem** [Xu et al. ICLR 2019]
  - GCN’s aggregation function cannot distinguish different multi-sets with the same color proportion.

**Failure case**

- **Why?**
For simplicity, we assume node colors are represented by **one-hot encoding**.

- **Example** If there are two distinct colors:

  \[
  \begin{align*}
  \text{\ding{51}} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \text{\ding{55}} &= \begin{pmatrix} 0 \\ 1 \end{pmatrix}
  \end{align*}
  \]

- This assumption is sufficient to illustrate how GCN fails.
**Neighbor Aggregation: Case Study**

- **GCN (mean-pool)** [Kipf & Welling ICLR 2017]
  - **Failure case illustration**

```
Linear + ReLU
(0.5)
(0.5)
```

*Element-wise-mean-pool*

```
(1) (0)
(0) (1)
```

```
Linear + ReLU
(0.5)
(0.5)
```

```
(1) (1) (0) (0)
(0) (0) (1) (1)
```

Same outputs!
**Neighbor Aggregation: Case Study**

- **GraphSAGE (max-pool)** [Hamilton et al. NeurIPS 2017]
  - Apply an MLP, then take **element-wise max**.
  - **Theorem** [Xu et al. ICLR 2019]
    - GraphSAGE’s aggregation **function cannot distinguish different multi-sets with the same set of distinct colors**.

**Failure case**

- **Why?**
GraphSAGE (max-pool) [Hamilton et al. NeurIPS 2017]

Failure case illustration

The same outputs!

Element-wise-max-pool

For simplicity, assume the one-hot encoding after MLP.
We analyzed the expressive power of GNNs.

Main takeaways:

- Expressive power of GNNs can be characterized by that of the neighbor aggregation function.
- Neighbor aggregation is a function over multi-sets (sets with repeating elements).
- GCN and GraphSAGE’s aggregation functions fail to distinguish some basic multi-sets; hence not injective.
- Therefore, GCN and GraphSAGE are not maximally powerful GNNs.
Our goal: Design maximally powerful GNNs in the class of message-passing GNNs.

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

Here, we design a neural network that can model injective multiset function.
**Theorem** [Xu et al. ICLR 2019]

Any injective multi-set function can be expressed as:

$$\Phi \left( \sum_{x \in S} f(x) \right)$$

Some non-linear function

**S : multi-set**

$$\begin{array}{c}
\text{Some non-linear function} \\
\text{Sum over multi-set}
\end{array}$$
Proof Intuition: [Xu et al. ICLR 2019]

$f$ produces one-hot encodings of colors. Summation of the one-hot encodings retains all the information about the input multi-set.

$$\Phi \left( \sum_{x \in S} f(x) \right)$$

Example:

$$\Phi \left( \begin{array}{c} f(\text{\large{\color{yellow}1}}) + f(\text{\large{\color{blue}0}}) + f(\text{\large{\color{blue}0}}) \end{array} \right)$$

One-hot

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$
Universal Approximation Theorem

- How to model $\Phi$ and $f$ in $\Phi(\sum_{x \in S} f(x))$?
- We use a Multi-Layer Perceptron (MLP).
- **Theorem: Universal Approximation Theorem**
  [Hornik et al., 1989]
  - 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.
Injective Multi-Set Function

- We have arrived at a neural network that can model any injective multiset 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.
Most Expressive GNN

- **Graph Isomorphism Network (GIN)** [Xu et al. ICLR 2019]
  - Apply an MLP, element-wise sum, followed by another MLP.
  
  \[ \text{MLP}_\Phi \left( \sum_{x \in S} \text{MLP}_f(x) \right) \]

- **Theorem** [Xu et al. ICLR 2019]
  - GIN‘s neighbor aggregation function is injective.
  - **No failure cases!**
  - **GIN is THE most expressive GNN in the class of message-passing GNNs!**
Full Model of GIN

- So far: We have described the neighbor aggregation part of GIN.

- We now describe the full model of GIN by relating it to **WL graph kernel** (traditional way of obtaining graph-level features).
  - We will see how GIN is a “neural network” version of the WL graph kernel.
Recall: Color refinement algorithm in WL kernel.

- **Given**: A graph $G$ with a set of nodes $V$.
  - Assign an initial color $c^{(0)}(v)$ 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.
Color Refinement (1)

Example of color refinement given two graphs

- Assign initial colors

  ![Initial Color Assignment](image)

- Aggregate neighboring colors

  ![Color Aggregation](image)
Example of color refinement given two graphs

- **Aggregated colors:**

- **Injectively** HASH the aggregated colors

**HASH table:** Injective!

<table>
<thead>
<tr>
<th>Color Pattern</th>
<th>HASH Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1</td>
<td>2</td>
</tr>
<tr>
<td>1,11</td>
<td>3</td>
</tr>
<tr>
<td>1,111</td>
<td>4</td>
</tr>
<tr>
<td>1,1111</td>
<td>5</td>
</tr>
</tbody>
</table>
Example of color refinement given two graphs

- Process continues until a stable coloring is reached
- Two graphs are considered **isomorphic** if they have the same set of colors.
GIN uses a neural network to model the injective HASH function.

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

Specifically, we will model the injective function over the tuple:

\[ (c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)}) \]

- Root node features
- Neighboring node colors
The Complete GIN Model

**Theorem** (Xu et al. ICLR 2019)

Any injective function over the tuple

\[
\begin{align*}
(c^{(k)}(v), \{c^{(k)}(u)\}_{u \in N(v)})
\end{align*}
\]

Root node feature

Neighboring node features

can be modeled as

\[
\text{MLP}_\Phi \left( (1 + \epsilon) \cdot \text{MLP}_f(c^{(k)}(v)) + \sum_{u \in N(v)} \text{MLP}_f(c^{(k)}(u)) \right)
\]

where \(\epsilon\) is a learnable scalar.
The Complete GIN Model

- If input feature \( c^{(0)}(v) \) is represented as one-hot, **direct summation is injective.**

Example:

\[
\Phi \left( \begin{array}{ccc}
0 & + & 0 \\
1 & + & 1 \\
0 & + & 1 \\
\end{array} \right) = \begin{array}{c}
1 \\
2 \\
\end{array}
\]

- We only need \( \Phi \) to ensure the injectivity.

\[
\text{GINConv} \left( c^{(k)}(v), \left\{ c^{(k)}(u) \right\}_{u \in N(v)} \right) = \text{MLP}_\Phi \left( (1 + \epsilon) \cdot c^{(k)}(v) + \sum_{u \in N(v)} c^{(k)}(u) \right)
\]

This MLP can provide “one-hot” input feature for the next layer.
GIN’s node embedding updates

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

- Assign an initial vector $c^{(0)}(v)$ to each node $v$.
- Iteratively update node vectors by

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

Differentiable color HASH function

where $\text{GINConv}$ maps different inputs to different embeddings.

- After $K$ steps of GIN iterations, $c^{(K)}(v)$ summarizes the structure of $K$-hop neighborhood.
GIN and WL Graph Kernel

- GIN can be understood as differentiable neural version of the WL graph Kernel:

<table>
<thead>
<tr>
<th></th>
<th>Update target</th>
<th>Update function</th>
</tr>
</thead>
<tbody>
<tr>
<td>WL Graph Kernel</td>
<td>Node colors (one-hot)</td>
<td>HASH</td>
</tr>
<tr>
<td>GIN</td>
<td>Node embeddings (low-dim vectors)</td>
<td>GINConv</td>
</tr>
</tbody>
</table>

- Advantages of GIN over the WL graph kernel are:
  - Node embeddings are **low-dimensional**; hence, they can capture the fine-grained similarity of different nodes.
  - Parameters of the update function can be **learned for the downstream tasks**.
Expressive Power of GIN

- Because of the relation between GIN and the WL graph kernel, their expressive is exactly the same.
  - If two graphs can be distinguished by GIN, they can be also distinguished by the WL kernel, and vice versa.

- How powerful is this?
  - WL kernel has been both theoretically and empirically shown to distinguish most of the real-world graphs [Cai et al. 1992].
  - Hence, GIN is also powerful enough to distinguish most of the real graphs!
We design a neural network that can model injective multi-set function.

We use the neural network for neighbor aggregation function and arrive at **GIN**—the most expressive GNN model.

The key is to use **element-wise sum pooling**, instead of mean-/max-pooling.

**GIN** is closely related to the WL graph kernel.

Both **GIN** and WL graph kernel can distinguish most of the real graphs!
The Power of Pooling

Failure cases for mean and max pooling:

(a) Mean and Max both fail
(b) Max fails
(c) Mean and Max both fail

Ranking by discriminative power:

Input | sum - multiset | mean - distribution | max - set
Can expressive power of GNNs be improved?

- There are basic graph structures that existing GNN framework cannot distinguish, such as difference in cycles.

- GNNs’ expressive power **can be improved** to resolve the above problem. [You et al. AAAI 2021, Li et al. NeurIPS 2020]