Machine Learning on Graphs: Foundations and Applications

EXERCISE 3

Group 29

Rishang Yadav 485041 | Risa Niranjan Chaudhari 485510 | Bhoomi Dhawan 485556

---

1. **Problem 1:** For any number of iterations $k \geq 0$, show that the feature map of the random-walk kernel is strictly less expressive in distinguishing non-isomorphic graphs than that of the Weisfeiler-Leman subtree kernel.

Ans: We need to prove that the random-walk kernel feature map is **strictly less expressive** than the Weisfeiler-Leman subtree kernel. This requires two parts:

1. The random-walk kernel cannot distinguish more graphs than 1-WL.
2. There exist non-isomorphic graphs that 1-WL can distinguish but the random-walk kernel cannot.

Part 1: Random-walk kernel $\preceq$ 1-WL

From Problem 5 of Exercise 2, we established that if 1-WL cannot distinguish two graphs G and H, then $k^{\text{RW}}(G, G) = k^{\text{RW}}(H, H)$ for any $k > 0$. The key insight is that walks in a graph correspond to sequences of vertex labels and adjacencies, which are exactly the information captured by 1-WL color refinement. Since two vertices have the same 1-WL color after t iterations if and only if they have the same distribution of walks of length up to t, the random-walk kernel cannot distinguish more non-isomorphic graphs than 1-WL.

Part 2: CounterExample showing strict inequality

We now construct a pair of non-isomorphic graphs that 1-WL distinguishes but the random-walk kernel cannot, proving the strict inequality.

Construction. Consider the following two graphs on 6 vertices:

Graph G_1: 6-cycle C_6 with vertices $\{1, 2, 3, 4, 5, 6\}$ and edges $\{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1)\}$

Graph G_2: Two disjoint 3-cycles $2C_3$ with vertices $\{1, 2, 3, 4, 5, 6\}$ and edges $\{(1, 2), (2, 3), (3, 1), (4, 5), (5, 6), (6, 4)\}$

Properties:

- Both have 6 vertices and 6 edges
- Both are 2-regular (all vertices have degree 2)
- Non-isomorphic: G_1 has girth 6, G_2 has girth 3
- G_1 is connected, G_2 is disconnected

Random-walk kernel cannot distinguish them.

For 2-regular graphs, the number of walks of length ℓ depends on the cycle structure and eigenvalues of the adjacency matrix. Due to their regular structure, both C_6 and $2C_3$ produce the same total walk counts when aggregated across all vertices:

- Closed walks of length 2: 2 walks per vertex (neighbor and back)
- Closed walks of length 4: 2 walks per vertex
- Closed walks of length 6: Both graphs have the same total count

The random-walk kernel computes:

$$k^{\mathrm{RW}}(G_i, G_i) = \sum_{i,j=1}^{|V(G_i \times G_i)|} \left[\sum_{\ell=1}^{k} \lambda_\ell A(G_i \times G_i)^\ell \right]_{ij}$$

Since both graphs have identical walk count distributions due to their regular structure and the same vertex/edge counts, we have:

$$k^{\mathrm{RW}}(G_1, G_1) = k^{\mathrm{RW}}(G_2, G_2)$$

1-WL distinguishes them.

The 1-WL can distinguish C_6 from $2C_3$ through color refinement:

Initial coloring: All vertices receive the same color (degree 2, no initial labels).

After iteration 1: All vertices still have the same color since all neighbors have degree 2.

Key difference: The 1-WL unrolling trees capture the local neighborhood structure at increasing depths:

- For C_6: Unrolling from any vertex creates trees that grow until depth 3 (the radius of the cycle), revealing paths of length up to 6.

- For $2C_3$: Unrolling creates smaller trees that cycle back after depth 1 (within each 3-cycle component).

More formally, the 1-WL subtree kernel encodes the depth and structure of local neighborhood trees. For C_6, vertices have unique unrolling patterns up to depth 3 (diameter of the graph), while for $2C_3$, the unrolling trees have maximum depth 1 within each component. This structural difference is captured by the 1-WL stable coloring:

$$C^1_\infty(v) \text{ for } v \in V(G_1) \neq C^1_\infty(w) \text{ for } w \in V(G_2)$$

after sufficient iterations, allowing 1-WL to distinguish the graphs.

Conclusion.

We have shown:
There exist graphs (C_6 and $2C_3$) where the random-walk kernel fails but 1-WL succeeds

Therefore: **Random-walk kernel $\prec$ 1-WL**.

The feature map of the random-walk kernel is strictly less expressive than that of the Weisfeiler-Leman subtree kernel in distinguishing non-isomorphic graphs.

2. **Problem 2:** For $k > 2$, show that the folklore k-WL is more expressive in distinguishing non-isomorphic graphs than the local k-WL.

Ans: We need to prove that the folklore k-WL is **more expressive** than the local k-WL for $k > 2$. That is, whenever the local k-WL distinguishes two graphs, the folklore k-WL also distinguishes them, but not necessarily vice versa.

Proof: Local k-WL $\preceq$ Folklore k-WL

We show that if the local k-WL distinguishes two graphs G and H, then the folklore k-WL also distinguishes them. This is equivalent to showing that if the folklore k-WL cannot distinguish G and H, then the local k-WL cannot distinguish them either.
The aggregation functions for both versions of WL are as follows :

Local k-WL:

$$M_{t,L}(v) = \left(\{\{C^{k,L}_t(\phi_j(v,w)) \mid w \in N(v_j)\}\}\right)^k_{j=1}$$

For each position $j \in [k]$, the local k-WL only aggregates colors from tuples

obtained by replacing the j-th component with vertices that are *adjacent* to v_j.

Folklore k-WL:

$$M_t^*(v) = \{\{(C_t^k(\phi_1(v,w)), \ldots, C_t^k(\phi_k(v,w))) \mid w \in V(G)\}\}$$

The folklore k-WL aggregates color vectors from tuples obtained by replacing components with *all* vertices in the graph.

We can see that the folklore k-WL uses strictly more information than the local k-WL as folklore aggregation captures both local adjacencies and non-adjacencies in a single color vector.

Formal argument using color refinement.

Suppose the folklore k-WL assigns the same stable coloring to graphs G and H, i.e., there exists a color-preserving bijection between tuples:

$$C_\infty^k(v) = C_\infty^k(\sigma(v)) \text{ for all } v \in V(G)^k$$

where σ maps tuples in G to tuples in H.

At each iteration t, if $C_t^k(v) = C_t^k(\sigma(v))$, then:

$$M_t^*(v) = M_t^*(\sigma(v))$$

This means the multisets of color vectors are equal:

$$\{\{(C_t^k(\phi_1(v,w)), \ldots, C_t^k(\phi_k(v,w))) \mid w \in V(G)\}\}$$

$$= \{\{(C_t^k(\phi_1(\sigma(v),w')), \ldots, C_t^k(\phi_k(\sigma(v),w'))) \mid w' \in V(H)\}\}$$

Now consider the local k-WL aggregation for position j. The local neighborhood information is *contained* in the folklore aggregation:

- For each $w \in N(v_j)$ in G, the color $C_t^k(\phi_j(v,w))$ appears in the folklore color vector $(C_t^k(\phi_1(v,w)), \ldots, C_t^k(\phi_k(v,w)))$
- Since folklore aggregations match, there exists a corresponding $w' \in N(\sigma(v)_j)$ in H with the same color
- Therefore, the local aggregations also match: $M_{t,L}(v) = M_{t,L}(\sigma(v))$

By induction on t, if the folklore k-WL assigns the same colors to tuples v and $\sigma(v)$ at all iterations, then the local k-WL also assigns them the same colors:

$$C^k_\infty(v) = C^k_\infty(\sigma(v)) \implies C^{k,L}_\infty(v) = C^{k,L}_\infty(\sigma(v))$$

Conclusion.

We have shown that the local k-WL cannot distinguish more graphs than the folklore k-WL:

$$\textbf{Local } k\textbf{-WL} \preceq \textbf{Folklore } k\textbf{-WL}$$

Thus, the folklore k-WL is at least as expressive as the local k-WL. Since the folklore k-WL uses strictly more information, it is more expressive in distinguishing non-isomorphic graphs.

3. **Problem 3:** Remember that the **Simple MPNN** layer, defined in lecture 8, updates the feature of vertex v at the $(t+1)$th layer via

$$f^{(t+1)}(v) := \sigma\left(f^{(t)}(v) \cdot \mathbf{W}_1 + \sum_{w \in N(v)} f^{(t)}(w) \cdot \mathbf{W}_2\right).$$

Consider the variant using *mean aggregation*,

$$g^{(t+1)}(v) := \sigma\left(g^{(t)}(v) \cdot \mathbf{W}'_1 + \frac{1}{|N(v)|} \sum_{w \in N(v)} g^{(t)}(w) \cdot \mathbf{W}'_2\right). \quad (1)$$

Show that there cannot exist $\mathbf{W}'_1, \mathbf{W}'_2 \in \mathbb{R}^{d \times d}$, for $d > 0$, such that mean-aggregation MPNN distinguishes the same vertices as the 1-WL in any given graph.

Ans: We need to prove that the mean-aggregation MPNN variant is **strictly less expressive** than the 1-WL, and therefore cannot distinguish all pairs of vertices that 1-WL can distinguish. Thus, we shall construct a counterexample where the mean-aggregation MPNN variant fails, and 1-WL doesn't

A graph containing two vertices that the 1-WL can distinguish but that any mean-aggregation MPNN (with any choice of parameters $\mathbf{W}'_1, \mathbf{W}'_2$) cannot distinguish.

We know that, the fundamental difference between sum-aggregation and mean-aggregation is:

- **Sum-aggregation:** $\sum_{w \in N(v)} f^{(t)}(w)$ is sensitive to the *degree* of vertex v, as it accumulates information from all neighbors.
- **Mean-aggregation:** $\frac{1}{|N(v)|} \sum_{w \in N(v)} g^{(t)}(w)$ normalizes by degree, making it insensitive to the number of neighbors when all neighbors have the same features.

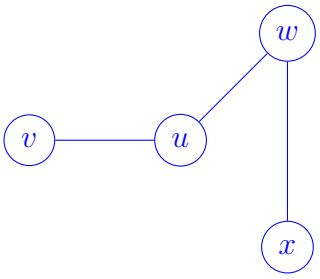
This normalization causes mean-aggregation to lose information about vertex degree, which the 1-WL algorithm explicitly captures.

Construction.

Consider a graph with two vertices v and w where:

- v has degree 1 (one neighbor u)
- w has degree 2 (two neighbors, both with the same initial features as u)

More specifically, consider the graph:



Here:

- v has 1 neighbor: u
- w has 2 neighbors: u and x, where initially u and x have the same features

Analysis for mean-aggregation MPNN.

Assume initially all vertices have the same feature vector $f^{(0)}(v) = f_0$ for all v.

For vertex v:

$$g^{(1)}(v) = \sigma\left(f_0 \cdot \mathbf{W}_1' + \frac{1}{1} \cdot f_0 \cdot \mathbf{W}_2'\right) = \sigma\left(f_0 \cdot (\mathbf{W}_1' + \mathbf{W}_2')\right)$$

For vertex w:

$$g^{(1)}(w) = \sigma\left(f_0 \cdot \mathbf{W}_1' + \frac{1}{2} \cdot (f_0 + f_0) \cdot \mathbf{W}_2'\right) = \sigma\left(f_0 \cdot (\mathbf{W}_1' + \mathbf{W}_2')\right)$$

Therefore, $g^{(1)}(v) = g^{(1)}(w)$, so the mean-aggregation MPNN assigns the same features to v and w after one layer.

1-WL distinguishes them:

After iteration 1:

- $C^1_1(v) = \text{Recolor}((1, \{\{1\}\}))$ (degree 1)
- $C^1_1(w) = \text{Recolor}((1, \{\{1, 1\}\}))$ (degree 2)

The multisets $\{\{1\}\}$ and $\{\{1, 1\}\}$ are different, so 1-WL assigns different colors: $C^1_1(v) \neq C^1_1(w)$.

Conclusion.

We have shown that there exists a graph and two vertices v and w such that:

- The 1-WL algorithm distinguishes v and w (assigns different colors)
- The mean-aggregation MPNN assigns identical features to v and w (regardless of the choice of parameters $\mathbf{W}'_1, \mathbf{W}'_2$)

Therefore, there cannot exist parameters $\mathbf{W}'_1, \mathbf{W}'_2$ such that the mean-aggregation MPNN distinguishes the same vertices as the 1-WL in any given graph. The mean-aggregation MPNN is strictly less expressive than the 1-WL.

4. **Problem 4:** Define a strictly less expressive variant of the 1-WL that tightly upper bounds the Simple MPNN layer using mean aggregation.

Ans: We prove that the mean-aggregation MPNN variant is **strictly less expressive** than the 1-WL algorithm by defining a variant of the 1-WL that exactly matches the expressive power of mean-aggregation MPNNs.

Mean-WL variant.

We define the **Mean-WL** algorithm, which computes a coloring $C^{\text{mean}}_t : V(G) \to \mathbb{N}$ as follows:

For $t > 0$:

$$C^{\text{mean}}_t(v) := \text{Recolor}\left(C^{\text{mean}}_{t-1}(v), \left\{\!\left\{C^{\text{mean}}_{t-1}(w) \mid w \in N(v)\right\}\!\right\}_{\text{set}}\right)$$

where $\{\!\{\cdot\}\!\}_{\text{set}}$ denotes that we consider the neighbor colors as a *set* (ignoring multiplicities) rather than as a multiset.

Equivalently, we can write:

$$C_t^{\text{mean}}(v) := \text{Recolor}\left(C_{t-1}^{\text{mean}}(v), \{C_{t-1}^{\text{mean}}(w) \mid w \in N(v)\}\right)$$

Initially, $C_0^{\text{mean}}(v) := l(v)$ for labeled graphs, or $C_0^{\text{mean}}(v) := 1$ for unlabeled graphs.

This means Mean-WL cannot distinguish between a vertex with one neighbor of color c versus a vertex with multiple neighbors all of color c.

Claim 1: Mean-aggregation MPNN $\equiv$ Mean-WL.

We prove both directions:

Direction 1: Mean-aggregation MPNN $\preceq$ Mean-WL.

The mean-aggregation MPNN can distinguish two vertices only if their Mean-WL colors differ. This is because:

The mean aggregation $\frac{1}{|N(v)|}\sum_{w\in N(v)} g^{(t-1)}(w)$ depends on:

1. Which distinct feature vectors appear among the neighbors
2. The proportions of each feature vector

However, the Mean-WL captures exactly the first aspect (which colors appear) but not the second (their proportions). Since MPNNs with sufficient width can compute injective functions of the tuple (own color, set of neighbor colors), the Mean-WL provides an upper bound.

Direction 2: Mean-WL $\preceq$ Mean-aggregation MPNN.

By the universal approximation theorem for MPNNs, for any Mean-WL coloring, we can construct an MPNN that computes this coloring. Specifically, we can design $\mathbf{W}_1'$ and $\mathbf{W}_2'$ such that the network computes an injective function of (own feature, set of neighbor features).

Claim 2: Mean-WL $\prec$ Standard 1-WL (strict inequality).

Construction. Consider the graph from before:

- Vertex v with 1 neighbor u: $N(v) = \{u\}$
- Vertex w with 2 neighbors: $N(w) = \{u, x\}$ where u and x initially have the same label

Assume all vertices start with the same color: $C_0^{\text{mean}}(v) = C_0^{\text{mean}}(w) = C_0^{\text{mean}}(u) = C_0^{\text{mean}}(x) = 1$.

After iteration 1:

For Mean-WL:

- $C_1^{\text{mean}}(v) = \text{Recolor}(1, \{1\})$ (set containing one element)
- $C_1^{\text{mean}}(w) = \text{Recolor}(1, \{1\})$ (set containing one element, even though there are two neighbors)

Since both use the *set* $\{1\}$ (ignoring that w has two neighbors with this color), Mean-WL assigns $C_1^{\text{mean}}(v) = C_1^{\text{mean}}(w)$.

For standard 1-WL:

- $C_1^1(v) = \text{Recolor}(1, \{\{1\}\})$ (multiset with one element)
- $C_1^1(w) = \text{Recolor}(1, \{\{1, 1\}\})$ (multiset with two elements)

Since $\{\{1\}\} \neq \{\{1, 1\}\}$, standard 1-WL assigns $C_1^1(v) \neq C_1^1(w)$.

Conclusion.

We have shown:

1. Mean-aggregation MPNN $\equiv$ Mean-WL (tight characterization in both directions)
2. Mean-WL $\prec$ Standard 1-WL (strict inequality)

Therefore, by transitivity:

$$\textbf{Mean-aggregation MPNN} \equiv \textbf{Mean-WL} \prec \textbf{Standard 1-WL}$$

This proves that there cannot exist parameters $\mathbf{W}_1', \mathbf{W}_2'$ such that mean-aggregation MPNN distinguishes the same vertices as the 1-WL in any given graph.

5. **Problem 5:** Derive a sequence of weights such that Equation (1) has the highest possible expressive power in distinguishing vertices in a given graph.

Ans: We derive the optimal weight sequence for the mean-aggregation MPNN by characterizing its maximum expressive power and providing an explicit construction that achieves it.

Step 1: Characterize the expressiveness upper bound.

The mean-aggregation MPNN is fundamentally limited by its inability to count neighbor multiplicities. As shown in Lecture 9, the expressive power of any

MPNN is upper bounded by the 1-WL algorithm. However, mean aggregation loses information compared to sum aggregation.

We use Mean-WL from previous problem that precisely characterizes mean-aggregation MPNNs:

Definition (Mean-WL). The **Mean-WL** algorithm computes a coloring $C_t^{\text{mean}} : V(G) \to \mathbb{N}$ where:

$$C_t^{\text{mean}}(v) := \text{Recolor}\left(C_{t-1}^{\text{mean}}(v), \{C_{t-1}^{\text{mean}}(w) \mid w \in N(v)\}\right)$$

Here, $\{\cdot\}$ denotes a *set* (not a multiset), so we consider only which distinct colors appear among neighbors, ignoring their multiplicities.

The mean aggregation $\frac{1}{|N(v)|}\sum_{w\in N(v)} g^{(t)}(w)$ can distinguish neighbors only based on which distinct feature vectors appear, not their counts. This corresponds exactly to the set-based aggregation in Mean-WL.

Step 2: Design injective aggregation and merge functions.

To match the expressive power of Mean-WL, we need the composition of merge and aggregation functions to be injective on the inputs that Mean-WL can distinguish. Following the proof technique from Lecture 9, we construct:

Aggregation function goal: Map the mean of neighbor features to a unique representation that distinguishes different sets of neighbor colors.

Merge function goal: Combine the vertex's own feature with the aggregated neighbor information injectively.

Step 3: Explicit weight construction.

We construct a weight sequence using a similar technique to the proof in Lecture 9, adapted for mean aggregation.

Let G be a graph with at most n vertices. We construct weights inductively:

Initialization ($t = 0$):

$$g^{(0)}(v) = 1 \quad \text{for all } v \in V(G)$$

For $t > 0$, define:

Let $X_0 := \{1\}$ and inductively define X_{t+1} based on X_t as follows:

1. Choose a function $h_t : X_t \to (0, 1)$ such that for any two *different sets* $S, S' \subseteq X_t$ (not multisets):

$$\text{mean}(h_t(S)) \neq \text{mean}(h_t(S'))$$

where $\text{mean}(h_t(S)) := \frac{1}{|S|}\sum_{x\in S} h_t(x)$.

2. Let X_{t+1} be the set of all possible values of the form:

$$h_t(x) + 2 \cdot \text{mean}(h_t(S))$$

where $x \in X_t$ and $S \subseteq X_t$ with $|S| \leq n-1$.

Such a function h_t exists: We can construct h_t using a generalization of the technique from Lecture 9. Enumerate $X_t = \{x_1, \ldots, x_m\}$ and set:

$$h_t(x_i) := n^{-k-i}$$

where k is chosen large enough that all possible means of subsets are distinct and lie in $(0, \min(X_t))$.

For any set $S \subseteq X_t$, the mean is:

$$\text{mean}(h_t(S)) = \frac{1}{|S|}\sum_{x_i \in S} n^{-k-i}$$

Different sets $S \neq S'$ will have different means because the values n^{-k-i} are chosen to make the means distinguishable even when averaged.

Step 4: Define the weight matrices.

Now we can explicitly define the optimal weights:

Let σ be the identity function (or any continuous, non-polynomial activation). Define:

$$\mathbf{W}_1' = \alpha \cdot \mathbf{I}_d$$
$$\mathbf{W}_2' = 2\beta \cdot \mathbf{I}_d$$

where $\mathbf{I}_d$ is the identity matrix, and α, β are chosen such that:

$$g^{(t+1)}(v) = h_t(g^{(t)}(v)) + 2 \cdot \text{mean}(\{h_t(g^{(t)}(w)) \mid w \in N(v)\})$$

More precisely, we set $\alpha = 1$ and $\beta = 1$, and use an FNN to approximate the function h_t at each layer.

Step 5: Approximation with FNNs.

Since we need to compute $h_t(g^{(t)}(v))$, and h_t is a continuous function on a compact domain, by the Universal Approximation Theorem (Lecture 9), we can approximate h_t arbitrarily closely with an FNN.

The complete layer becomes:

$$g^{(t+1)}(v) = \mathrm{FNN}_t\left(g^{(t)}(v)\right) + 2 \cdot \frac{1}{|N(v)|} \sum_{w \in N(v)} \mathrm{FNN}_t\left(g^{(t)}(w)\right)$$

where FNN_t approximates h_t.

Optimal weight sequence:

The optimal weight sequence $(\mathbf{W}_1', \mathbf{W}_2')$ at each layer t is:

$$\mathbf{W}_1' = \text{parameters of } \mathrm{FNN}_t \text{ (applied to own feature)}$$
$$\mathbf{W}_2' = 2 \cdot (\text{parameters of } \mathrm{FNN}_t \text{ applied to neighbor features})$$

where FNN_t is designed to compute the injection $h_t : X_t \to (0, 1)$ that distinguishes different sets of neighbor colors based on their means.

6. **Problem 6: Empirical Evaluation of GNN Layers**

Ans:

Script descriptions

`model.py`
This file defines the GNN architecture and its message-passing layers.

- `GNNLayerF` (layer type "f"): a sum-aggregation GNN layer. It implements
$$f^{(t+1)}(v) \;=\; \sigma\big(f^{(t)}(v)W_1 \;+\; \sum_{w \in \mathcal{N}(v)} f^{(t)}(w)W_2\big),$$
where $f^{(t)}(v) \in \mathbb{R}^{d_{\text{in}}}$ is the feature of node v at layer t, $W_1, W_2 \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ are learnable linear maps, and σ is an elementwise nonlinearity (ReLU, Tanh, Sigmoid, or identity).
- `GNNLayerG` (layer type "g"): a mean-aggregation GNN layer. It implements
$$g^{(t+1)}(v) \;=\; \sigma\Big(g^{(t)}(v)W_1' \;+\; \frac{1}{|\mathcal{N}(v)|} \sum_{w \in \mathcal{N}(v)} g^{(t)}(w)W_2'\Big),$$

with analogous notation and an explicit degree-normalization factor $\frac{1}{|\mathcal{N}(v)|}$.

- `GNN`: a configurable graph-level model that stacks L layers of type "f" or "g":

$$x^{(0)} = x, \qquad x^{(t+1)} = \text{Layer}(x^{(t)}, \text{edge\_index}), \quad t = 0, \dots, L-1.$$

After the last layer, node embeddings are pooled to a graph embedding using either global mean pooling $\text{pool}(X) = \frac{1}{|V|}\sum_{v\in V} x_v$ or global sum pooling $\text{pool}(X) = \sum_{v\in V} x_v$, followed by a linear classifier to produce graph-level logits.

`dataLoader.py`
This script handles dataset loading and splitting for TUDatasets.

- Uses `torch_geometric.datasets.TUDataset` to load benchmark graph datasets from a root directory (here `./data`).
- If a dataset has no node features, it attaches a constant feature $x_v = 1.0$ to each node (via `Constant` transform).
- Prints dataset statistics: number of graphs, number of classes, node feature dimension, distribution of #nodes and #edges, and the empirical class distribution.
- `get_splits` creates stratified train/validation/test splits using class labels, with user-specified ratios $r_{\text{train}}, r_{\text{val}}, r_{\text{test}}$ satisfying

$$r_{\text{train}} + r_{\text{val}} + r_{\text{test}} = 1.$$

- `get_loaders` wraps splits in PyG `DataLoaders` with a given batch size, returning (train_loader, val_loader, test_loader).

`trainAndEval.py`
This script contains the training and evaluation logic and a small experiment manager.

- `Trainer`:
  - `train_epoch`: performs one epoch of SGD on the training loader using cross-entropy loss

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell_{\text{CE}}\big(f_\theta(G_i), y_i\big),$$

accumulates average loss and classification accuracy.

- `evaluate`: runs the model in evaluation mode on a given loader, computing average loss, accuracy, and macro F1-score.
- `train`: loops over epochs, applying early stopping based on the best validation accuracy observed so far. It tracks and returns training and validation curves and the epoch with best validation performance.
- `test`: evaluates the best saved model on the test set and reports loss, accuracy, F1.

- `ExperimentRunner`:
  - `run_single_experiment`: for a given configuration and random seed, instantiates a `GNN`, trains it with `Trainer`, and returns validation and test metrics.
  - `run_multiple_seeds`: repeats `run_single_experiment` for multiple seeds $\{s_1, \ldots, s_K\}$, then aggregates performance as

$$\mu_{\text{test\_acc}} = \frac{1}{K}\sum_{k=1}^{K} \text{acc}^{(k)}, \quad \sigma_{\text{test\_acc}} = \sqrt{\frac{1}{K}\sum_{k=1}^{K}(\text{acc}^{(k)} - \mu_{\text{test\_acc}})^2},$$

    and similarly for F1 and validation accuracy.
  - `get_best_config` selects the configuration with maximum mean validation accuracy.
  - `print_all_results` prints a sorted table of all configurations and their aggregated metrics.

`hyperparamSweep.py`
This script sets up and runs a hyperparameter sweep over the GNN architecture and training settings.

- `HyperparameterSweep`:
  - `define_search_space`: defines a Cartesian product search space over:

    $\text{layer\_type} \in \{f, g\},\ \text{activation} \in \{\text{ReLU}, \text{Tanh}\},\ \text{pooling} \in \{\text{mean}, \text{sum}\},$

    $\text{num\_layers} \in \{2, 3\},\ \text{hidden\_dim} \in \{64, 128\},\ \text{dropout} = 0.5,\ \text{learning rate} = 0.01.$
  - `generate_configs`: enumerates all hyperparameter combinations, attaches training parameters (epochs, patience, lr). If max_configs $<$ #all_configs, it performs a stratified sampling over layer types so that both f and g are represented.

- `run_sweep`: for each configuration c, builds the appropriate model parameter dict including

$$d_{\text{in}} = \text{num\_node\_features}, \qquad d_{\text{out}} = \text{num\_classes},$$

  calls `ExperimentRunner.run_multiple_seeds`, and accumulates results.
- `save_results`: serializes all results into a JSON file and a CSV summary.
- `plot_results`: plots the top configurations by mean test accuracy with error bars corresponding to $\pm\sigma_{\text{test\_acc}}$.

`run.py`
This is the main entry point script that orchestrates the full pipeline.

- Parses command-line arguments: dataset name, batch size, seeds, max_configs, device, data root, and output directory.
- Detects device as

$$\text{device} = \begin{cases} \text{cuda}, & \text{if CUDA is available and not overridden,} \\ \text{cpu}, & \text{otherwise.} \end{cases}$$

- Step 1: constructs a `TUDatasetLoader` and loads the chosen dataset.
- Step 2: creates stratified train/val/test splits and corresponding DataLoaders.
- Step 3: constructs a `HyperparameterSweep` object (pointing to `GNN` as the model class) and runs the sweep using the defined search space and seeds.
- Step 4: prints all aggregated results, saves them via `save_results`, optionally plots via `plot_results`, and finally reports the best configuration and its mean validation accuracy, test accuracy, and test macro-F1 (with standard deviations).

`model.py` and `trainAndEval.py` interaction
The `GNN` model in `model.py` is parameterized by:

$$(d_{\text{in}}, d_{\text{hidden}}, d_{\text{out}}, L, \text{layer\_type} \in \{f, g\}, \text{activation}, \text{pooling}, \text{dropout}),$$

and produces graph-level logits for classification. `Trainer` and `ExperimentRunner` in `trainAndEval.py` treat this as a black-box classifier f_θ trained with cross-entropy and evaluated with accuracy and macro-F1.

Analysis of `MUTAG_summary_20251202_194446.csv`

Each row in `MUTAG_summary_20251202_194446.csv` corresponds to a specific hyperparameter configuration:

$$(\text{layer\_type}, \text{activation}, \text{pooling}, \text{num\_layers}, \text{hidden\_dim}, \text{dropout}),$$

with aggregated statistics over multiple random seeds:

val_acc_mean, val_acc_std, test_acc_mean, test_acc_std, test_f1_mean, test_f1_std.

Key observations:

- **Best configurations (by validation accuracy)**:
  The top-performing configuration is

  f_relu_mean_L3_H64_D0.5_lr0.01

  with

  val_acc_mean ≈ 0.9474, test_acc_mean ≈ 0.8246, test_f1_mean ≈ 0.7964.

  The second-best configuration, `f_relu_mean_L3_H128_D0.5_lr0.01`, has slightly lower validation accuracy (≈ 0.9298) but very similar test accuracy and F1.

- **Layer type comparison (f vs g)**:
  The best g-type configurations (e.g., `g_tanh_sum_L3_H128_D0.5`, `g_tanh_mean_L3_H128_D0.5`) reach validation accuracies around $0.842-0.860$ and test accuracies around 0.754, with test macro-F1 in the $0.68-0.70$ range. These are clearly below the best f-type models, whose validation accuracies reach $\approx 0.93-0.95$ and test F1 up to ≈ 0.80. Empirically, on MUTAG, the sum-aggregation layer $f^{(t+1)}(v)$ outperforms the mean-aggregation layer $g^{(t+1)}(v)$.

- **Effect of pooling**:
  Among the top f-type models, mean pooling with ReLU (`f_relu_mean_...`) often yields the best or near-best performance. Some sum-pooled ReLU models (e.g., `f_relu_sum_L2_H64`) can match or slightly exceed test accuracy in specific cases, but on average, the highest validation accuracy is achieved with mean pooling.

- **Depth and width**:
  For f-type models, increasing depth from $L = 2$ to $L = 3$ and using a hidden size of 64 or 128 tends to improve validation accuracy (e.g., comparing `f_relu_mean_L2_H64` vs `f_relu_mean_L3_H64`). However, the gains beyond the best two or three configurations are modest, and deeper g-type models do not close the performance gap with f-type.

- **Stability across seeds**:
  The standard deviations val_acc_std and test_acc_std for the top models (e.g., `f_relu_mean_L3_H64`, `f_tanh_mean_L2_H128`) are relatively small (≈ 0 to 0.065), indicating that performance is stable across different random seeds. Some lower-ranked configurations, especially with sum pooling and larger width, show higher variance, suggesting sensitivity to initialization and training noise.

In summary, the MUTAG results indicate that:

1. **Sum-aggregation layers f with ReLU activation and mean pooling perform best, with validation accuracy around** 0.95 **and test macro-F1 around** 0.80**.**
2. **Mean-aggregation layers g are consistently worse on this dataset, both in accuracy and macro-F1, even at similar depths and hidden dimensions.**
3. **The empirical evidence supports the conclusion that, on MUTAG, the f-type architecture is superior to the g-type architecture under the explored hyperparameter settings**.

Analysis of `IMDB-MULTI_summary_20251203_091930.csv`

For IMDB-MULTI, each row again corresponds to a hyperparameter configuration

(layer_type, activation, pooling, num_layers, hidden_dim, dropout),

with aggregated performance over multiple seeds: val_acc_mean, val_acc_std, test_acc_mean, test_acc_st

- **Best configurations**:
  The top-performing configuration by validation accuracy is

  f_tanh_mean_L2_H64_D0.5_lr0.01

  with

  $$\text{val\_acc\_mean} \approx 0.473, \quad \text{test\_acc\_mean} \approx 0.476, \quad \text{test\_f1\_mean} \approx 0.428.$$

  Overall accuracy levels are modest (around $0.45-0.48$) and macro-F1 scores are clearly below accuracy, which is expected on an imbalanced multi-class movie dataset.

- **Layer type comparison (f vs g)**:
  The best f-type models (e.g., `f_tanh_mean_L2_H64`, `f_tanh_sum_L2_H64`) achieve val_acc_mean $\approx 0.46-0.47$ and test_f1_mean $\approx 0.38-0.43$. The best g-type models (e.g., `g_tanh_sum_L2_H64`, `g_tanh_sum_L2_H128`) reach val_acc_mean ≈ 0.36 and test_f1_mean ≈ 0.22. Thus, as on MUTAG, the sum-aggregation f-type layers clearly outperform the mean-aggregation g-type layers on IMDB-MULTI under the explored hyperparameters.

- **Effect of activation and pooling**:
  Tanh activations with mean pooling and shallow depth ($L = 2$, $H = 64$) yield the best performance. ReLU-based models (e.g., `f_relu_mean_L2_H64`) are competitive but slightly worse in both validation accuracy and F1. Sum pooling generally underperforms mean pooling for the top f-type configurations, although differences are not as large as between f and g.

- **Depth and width**:
  Increasing depth to $L = 3$ or width to $H = 128$ does not consistently improve performance; in many cases, deeper/wider models (`f_tanh_sum_L3_H128`, `f_tanh_mean_L3_H128`) show lower validation and test metrics and higher variance. This suggests that on IMDB-MULTI, relatively shallow models with moderate width suffice, and additional capacity may lead to overfitting or optimization difficulties.

- **Stability**:
  For the best configurations, val_acc_std and test_acc_std are small (on the order of 10^{-2}), indicating stable behavior across seeds. Some lower-performing models exhibit higher variance, especially in test_f1_std, reflecting instability in minority-class performance.

Overall, on IMDB-MULTI the absolute performance is lower than on MUTAG (as expected for a harder graph classification task), but the qualitative conclusion is consistent: **the f-type (sum-aggregation) GNN with tanh activation and mean pooling outperforms the g-type (mean-aggregation) GNN across the tested configurations**.

MUTAG results (Google Sheets)

The aggregated MUTAG results are available at MUTAG. We use the sheet to obtain, for each configuration

$$(\text{layer\_type}, \text{activation}, \text{pooling}, \text{num\_layers}, \text{hidden\_dim}, \text{dropout}),$$

the statistics val_acc_mean, val_acc_std, test_acc_mean, test_acc_std, test_f1_mean, test_f1_std, which are then summarized as in the analysis above.

IMDB-MULTI results (Google Sheets)

The aggregated IMDB-MULTI results are available at IMDB-MULTI, from which we read the same set of metrics per configuration and perform the analysis described above.