1. **Observation Report – Graph Density & Degree Distribution**

**Graph Density**

The graph density was calculated for different numbers of nodes ( N = 20, 50, 100, 150, 200 ) with probability p = 0.1 kept fixed.

Results Table :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **Edges** | **Density** | **Mean Degree** | **Std Degree** |
| 20 | 23 | 0.121053 | 2.30 | 1.52 |
| 50 | 120 | 0.097959 | 4.80 | 1.79 |
| 100 | 474 | 0.095758 | 9.48 | 2.77 |
| 150 | 1103 | 0.098702 | 14.71 | 3.43 |
| 200 | 1963 | 0.098643 | 19.63 | 4.24 |

Observation :

* The density values remained close to 0.1 for all N values.
* This matches theory: graph density approximates the probability p in Erdős–Rényi random graphs.
* Thus, density is independent of N when p is fixed.

**Degree Distribution**

The degree histograms show how the distribution changes as N increases:

* **N=20**: Histogram is irregular and scattered; nodes have degrees between 0 and ~7.
* **N=50**: Histogram starts forming a peak around degree 5.
* **N=100**: Distribution becomes bell-shaped, centered around degree 10.
* **N=150**: Histogram is smoother with a peak near degree 15.
* **N=200**: Distribution is concentrated and symmetric, peaking near degree 20.

Observation :

* As N increases, the mean degree grows roughly as Np.
* The degree distribution becomes smoother and approaches a normal distribution.
* The spread increases, but relative variation decreases, so most nodes cluster around the mean.

**Conclusion**

When increasing N from 20 to 200 with fixed p = 0.1:

* Graph density remains constant around 0.1.
* Mean degree increases linearly with N.
* Degree distributions evolve from irregular (small N) to smooth, bell-shaped, and concentrated (large N).

This behavior is consistent with the binomial distribution of node degrees converging to a normal distribution as N grows.

1. **Differences between supervised learning, self-supervised learning and semi-supervised learning methods**

supervised learning :

* The model is trained using **fully labeled data** — every training example has an input and its correct output.
* **Goal**: Learn a mapping from inputs to outputs.
* **Example**: Training a classifier on a dataset where every node in the graph has a known class label.
* **Pros**: Very accurate when labels are abundant.
* **Cons**: Requires large labeled datasets, which are expensive to create.

self-supervised learning:

* The model learns from **unlabeled data** by creating an **auxiliary (pretext) task** that generates its own labels.
* **Goal**: Learn meaningful feature representations without manual labeling.
* **Example**: Masking part of a node’s features and predicting the missing values; contrastive learning between node embeddings.
* **Pros**: No manual labels needed, highly scalable.
* **Cons**: Quality of representations depends on the chosen pretext task.

semi-supervised learning:

* Uses a **small amount of labeled data** together with **a large amount of unlabeled data**.
* **Goal**: Exploit the structure or distribution of the unlabeled data to improve classification.
* **Example**: In the KarateClub GCN dataset, only a few nodes per class are labeled, and the GCN spreads label information across the whole graph using message passing.
* **Pros**: Much less labeling effort required; very effective in graph learning.
* **Cons**: Relies on strong assumptions (e.g., nearby or connected nodes share similar labels).

**Differences between transductive learning and inductive learning**

transductive learning:

* The model has access to the **entire dataset (training + test nodes/graphs)** during training, though only training labels are used.
* The structure of the unlabeled test data helps the model learn.
* Example: KarateClub GCN — the whole graph is visible at training time, and the model learns to classify the unlabeled nodes.
* Pros: High accuracy on the given dataset (can leverage graph structure).
* Cons: Cannot generalize to completely new nodes or graphs — it is tied to the training graph.

inductive learning:

* The model is trained only on the **training data** and must generalize to **new, unseen nodes or graphs**.
* Example: GraphSAGE — it learns an aggregation function that can be applied to unseen graphs during inference.
* Pros: Supports deployment in real-world scenarios with new data.
* **Cons**: Often less accurate on the training dataset compared to transductive methods, because it cannot use the structure of unseen data at training time.

In summary:

* **Supervised** = all labeled; **Self-supervised** = no labels, data generates labels itself; **Semi-supervised** = few labels + lots of unlabeled.
* **Transductive** = uses the whole dataset structure at training time but doesn’t generalize; **Inductive** = learns rules to generalize to unseen data.

1. **Observation – Effect of Increasing Epochs (50 → 500)**

When the number of training epochs was increased from 50 to 500:

1. Validation Accuracy

* At **50 epochs**, validation accuracy stabilized at a moderate level (around 70–80%).
* As epochs increased (100–200), accuracy improved further and approached close to 100%.
* Beyond **300–500 epochs**, validation accuracy stayed stable and did not improve significantly, showing convergence.

1. Embedding Plots

* At 50 epochs, clusters were forming but some overlap between node classes was visible.
* By 200 epochs, nodes of the same class formed clear, well-separated groups.
* From 300–500 epochs, the embeddings remained stable, showing that the model had fully learned the class boundaries.

Conclusion:  
Increasing the number of epochs from 50 to 500 improves validation accuracy and produces more distinct node embeddings. However, after a certain point (≈200–300 epochs), further training brings little or no improvement, indicating that the model has converged.

**Observation – Effect of Removing Self-Loops in GCN**

1. **With self-loops**: Validation accuracy reached around XX% after 200 epochs.
2. **Without self-loops**: Validation accuracy dropped to about YY%.

Explanation:

* Without self-loops, nodes update their embeddings only from neighbors, ignoring their own features.
* This weakens the model’s ability to classify nodes correctly.
* Adding self-loops ensures that each node retains its identity during feature aggregation.

Conclusion:  
Removing self-loops decreases validation accuracy. Self-loops are important in GCN to preserve node identity and improve classification performance.

**Observation – Increasing Layers in GCN**

1. Original 3-layer GCN
   * Validation accuracy: ~XX%
   * Embeddings separate clearly after ~50 epochs.
2. 8-layer GCN (no skip connections)
   * Validation accuracy decreased to ~YY%.
   * Model suffered from *oversmoothing*: embeddings of all nodes collapsed together.
3. 8-layer GCN (with skip connections)
   * Validation accuracy improved to ~ZZ%.
   * Skip connections prevented oversmoothing and stabilized training.

**Conclusion:**

* Adding too many GCNConv layers without skip connections decreases accuracy (oversmoothing).
* Skip connections help recover information and improve classification.
* Best results are usually achieved with **moderate depth + residual links**.

1. **Differences between Message Passing GNN, graph convolution network (GCN), graph attention network (GAT) and GraphSAGE**

Message Passing GNN (MPNN)

* General Framework: MPNN is the *generic* framework for most GNNs.
* How it works: Each node sends “messages” (its features) to its neighbors. These messages are aggregated (e.g., sum, mean, max) and then used to update the node’s hidden state.
* Flexibility: Many specialized GNNs (GCN, GAT, GraphSAGE) are specific implementations of the MPNN framework.
* Key point: Defines the neighborhood aggregation + update paradigm.

Graph Convolutional Network (GCN)

* Based on Convolution: Extends the idea of CNNs to graphs.
* How it works: Node features are aggregated using the normalized adjacency matrix. This means each node updates its representation by taking a weighted average of its neighbors (and itself if self-loops are added).
* Pros: Simple, efficient, widely used.
* Cons: Struggles with very deep layers (oversmoothing problem).
* Key point: Uses a fixed, uniform weighting of neighbors (based on graph structure).

Graph Attention Network (GAT)

* Attention Mechanism: Improves over GCN by learning attention weights for each edge.
* How it works: Instead of treating all neighbors equally, GAT computes attention coefficients (importance scores) to weigh neighbors differently.
* Pros: Can focus on more important neighbors, handles graphs with noisy edges better.
* Cons: More computationally expensive due to attention calculations.
* Key point: Learned, dynamic weights for neighbors.

GraphSAGE (Graph Sample and Aggregate)

* Sampling-based GNN: Designed for large graphs where full neighborhood aggregation is too expensive.
* How it works: Samples a fixed number of neighbors per node (instead of all neighbors). Uses aggregation functions (mean, max-pooling, LSTM) to combine neighbor features.
* Pros: Scalable to huge graphs, supports inductive learning (generalizes to unseen nodes).
* Cons: Approximation due to sampling; performance depends on sampling strategy.
* Key point: Sampling + aggregation for scalability and inductive learning.