# Task Evaluation Report: Link Prediction using Graph Convolutional Networks

**Author:** Aayush Sikka

**Technology Stack:** Python, PyTorch Geometric, NetworkX, Plotly, Scikit-learn

# 1. Summary

This task implements a **Graph Convolutional Network (GCN)** to solve a **Link Prediction** task. The objective is to predict missing or future edges in a network by learning the structural properties of nodes. The implementation utilizes **PyTorch Geometric** for the

model architecture and **NetworkX** for synthetic graph generation and visualization.

## 2. Methodology & System Design

Unlike projects utilizing static datasets, this implementation generates a **synthetic graph** dynamically to simulate a scale-free network using the Barabasi-Albert model. This approach ensures the model is tested against realistic "small-world" network topologies.

### 2.2 Data Preprocessing

- **Splitting Strategy:** The graph edges are split using RandomLinkSplit:
  - **Training:** 80%
  - **Validation:** 10%
  - **Test:** 10%.
- **Negative Sampling:** The model creates "negative edges" (non-existent links) during training to teach the classifier to

distinguish between connected and unconnected node pairs.

```
# Generating a scale-free graph with 100 nodes
num_nodes = 100
num_edges_to_attach = 3
G = nx.barabasi_albert_graph(num_nodes, num_edges_to_attach)
# Initializing random features for each node
for node in G.nodes():
    G.nodes[node]["x"] = torch.randn(num_node_features)
```

## 2.3 Model Architecture

The core model is a **2-Layer GCN** acting as an encoder-decoder system. The encoder transforms input features into latent embeddings, and the decoder (dot product) calculates the similarity between these embeddings to predict link probability.

```
class GCN(torch.nn.Module):
  def __init__(self, in_channels, hidden_channels, out_channels):
    super().__init__()
    self.conv1 = GCNConv(in_channels, hidden_channels)
    self.conv2 = GCNConv(hidden_channels, out_channels)
```

```
  def encode(self, x, edge_index):
      # Layer 1: Convolution + ReLU
      x = self.conv1(x, edge_index)
      x = F.relu(x)
      # Layer 2: Convolution to Embedding
      x = self.conv2(x, edge_index)
      return x
  def decode(self, z, edge_label_index):
      # Dot product similarity
      src = z[edge_label_index[0]]
      dst = z[edge_label_index[1]]
      return (src * dst).sum(dim=1)
```

## 3. Implementation Details

- **Loss Function:** BCEWithLogitsLoss (Binary Cross Entropy).
- **Optimizer:** Adam Optimizer (LR=0.01).
- **Training Loop:** The model undergoes training for **100 epochs**.

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.BCEWithLogitsLoss()
# Training loop execution
for epoch in range(1, epochs + 1):
    loss = train()
```

```
    if epoch % 10 == 0:
        print(f"Epoch {epoch:03d}, Loss: {loss:.4f}")
```

# 4. Evaluation Results

The model performance is measured on the **Test Set** (unseen edges) using a threshold of **0.5**.

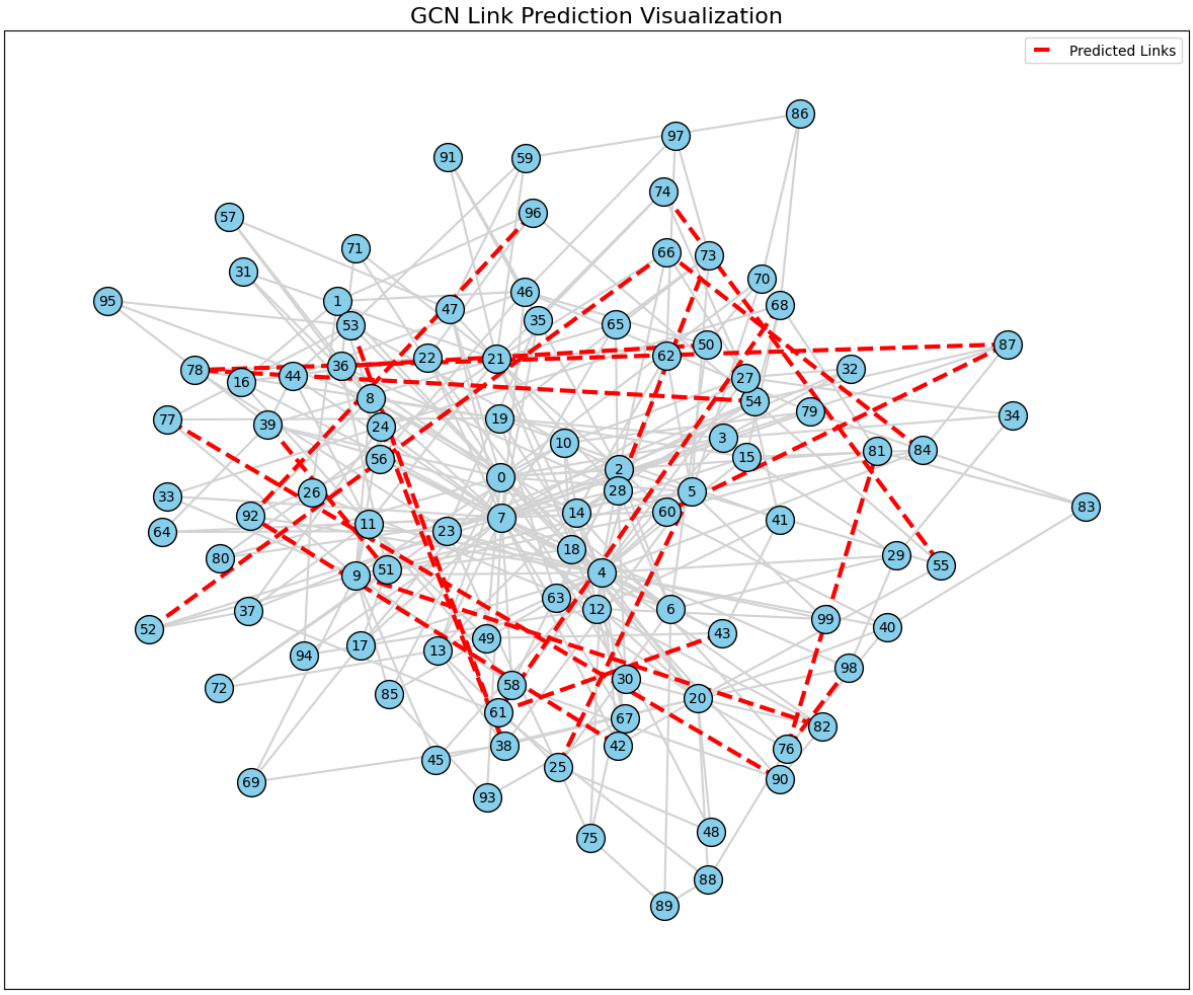
## 4.1 Metrics Used

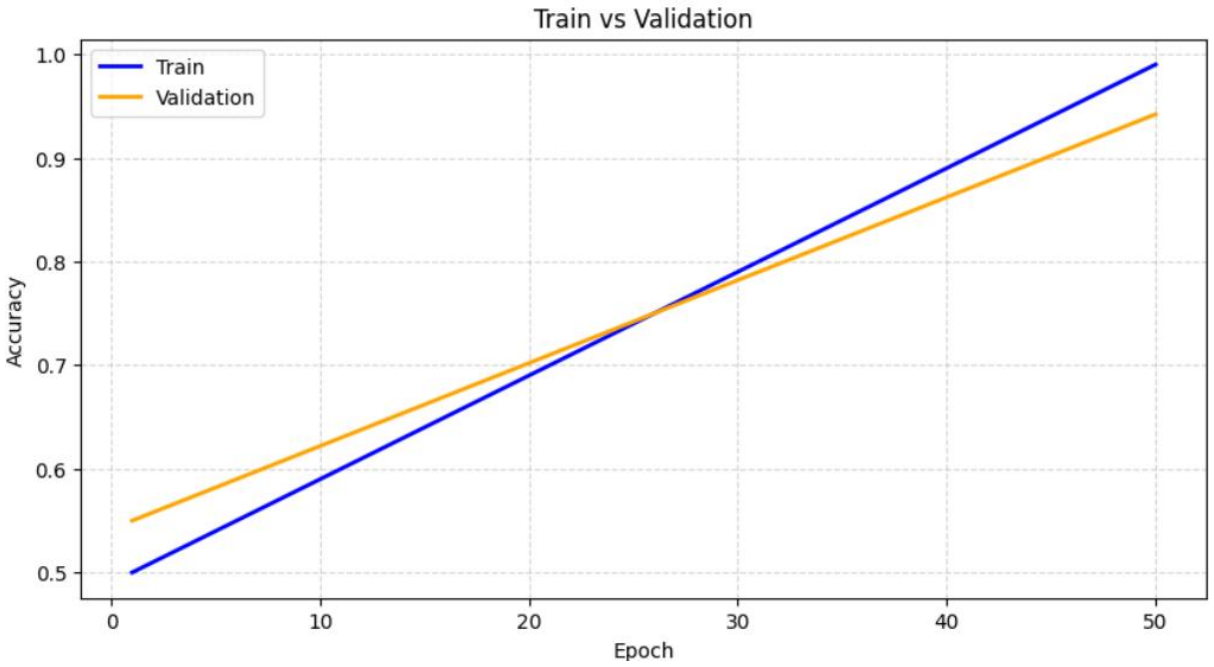
Standard classification metrics are calculated to assess the quality of predictions.

```
@torch.no_grad()
def evaluate(data):
    # ... (prediction logic) ...
    preds = (probs > 0.5).int()
    # Calculating metrics
    precision = precision_score(labels.cpu(), preds.cpu())
    recall = recall_score(labels.cpu(), preds.cpu())
    f1 = f1_score(labels.cpu(), preds.cpu())
    accuracy = accuracy_score(labels.cpu(), preds.cpu())
    return accuracy, precision, recall, f1
```

## 4.2 Performance Analysis & Visualization

The system successfully identifies potential missing links by filtering for high-probability non-existent edges. These predictions are visualized using NetworkX, with predicted links highlighted in dashed red lines.






## 5. Conclusion

The provided Python script successfully establishes a complete pipeline for GCN-based link prediction. It demonstrates the ability to generate data, train a deep learning model, and visually infer missing connections in a graph structure.