GraphSAGE Node Classification Report

Student Name: Yahya Ahmed Tawfik

Student ID: 2205126

Subject: Social Network Analysis

1. Environment Setup and Library Import

The initial phase of the project involves setting up the deep learning environment necessary for Graph Neural Networks (GNNs). We utilize **PyTorch Geometric (PyG)**, a library built upon PyTorch designed specifically for handling irregularly structured input data such as graphs.

We import SAGEConv, which implements the GraphSAGE operator. Unlike standard GCNs that require the entire graph during training (transductive), GraphSAGE learns to generate embeddings by sampling and aggregating features from a node's local neighborhood, making it scalable and suitable for inductive learning.

```
!pip install torch_geometric

import torch
from torch_geometric.data import Data
from torch_geometric.nn import SAGEConv
import torch.nn.functional as F
```

2. Feature Initialization (Node Attributes)

We define the feature matrix $x$, which serves as the initial state of the nodes. In this graph, we have **6 nodes**, each possessing **2 features**. The features are designed to distinguish between two hypothetical classes of users:

- **Benign Users (Nodes 0, 1, 2):** Characterized by the feature vector [1.0, 0.0].
- **Malicious Users (Nodes 3, 4, 5):** Characterized by the feature vector [0.0, 1.0].

This distinct initialization provides a clear ground truth for the model to learn from.

```
# Feature matrix x of shape [6, 2]
x = torch.tensor([
  [1.0, 0.0], # Node 0 (Benign)
  [1.0, 0.0], # Node 1 (Benign)
  [1.0, 0.0], # Node 2 (Benign)
  [0.0, 1.0], # Node 3 (Malicious)
  [0.0, 1.0], # Node 4 (Malicious)
```

```
    [0.0, 1.0]  # Node 5 (Malicious)
], dtype=torch.float)
```

3. Graph Topology Construction

The structure of the social network is defined using an edge index. The graph consists of two distinct communities (cliques) loosely connected by a single bridge.

- **Community A (Benign):** Nodes 0, 1, and 2 are fully connected to each other.
- **Community B (Malicious):** Nodes 3, 4, and 5 are fully connected to each other.
- **The Bridge (Cross-Edge):** An edge exists between Node 2 and Node 3. This is crucial as it represents the interaction between the two different classes.

The edge_index tensor defines the source and target nodes for every edge. Since the graph is undirected, every connection (u, v) must also have a corresponding (v, u) entry.

```
edge_index = torch.tensor([
    [0, 1], [1, 0],
    [1, 2], [2, 1],
    [0, 2], [2, 0],  # Benign Cluster
    [3, 4], [4, 3],
    [4, 5], [5, 4],
    [3, 5], [5, 3],  # Malicious Cluster
    [2, 3], [3, 2]   # Bridge Edge
], dtype=torch.long).t().contiguous()
```

4. Label Assignment

We define the ground truth labels y for supervised training. The task is **Binary Node Classification**:

- **Class 0:** Benign
- **Class 1:** Malicious

```
y = torch.tensor([0, 0, 0, 1, 1, 1])
```

5. Model Architecture: GraphSAGE

We construct a Graph Neural Network using two SAGEConv layers. The architecture follows this flow:

1. **First Convolution (conv1):** Takes the input dimensions (2 features) and maps them to a hidden dimension of 4. This layer aggregates information from immediate neighbors (1-hop).
2. **Activation Function (ReLU):** Introduces non-linearity to helping the model learn complex patterns.
3. **Second Convolution (conv2):** Maps the hidden dimension (4 features) to the output dimension (2 classes). This layer aggregates information from neighbors of neighbors (2-hops).
4. **Softmax Output:** Applies log_softmax to convert the raw logits into log-probabilities for classification.

```
class GraphSAGEModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        # Layer 1: Input 2 features -> Hidden 4 features
        self.conv1 = SAGEConv(2, 4)
        # Layer 2: Hidden 4 features -> Output 2 classes
        self.conv2 = SAGEConv(4, 2)

    def forward(self, x, edge_index):
        # First Message Passing Step
        x = self.conv1(x, edge_index)
        x = F.relu(x)

        # Second Message Passing Step
        x = self.conv2(x, edge_index)

        # Output Log-Probabilities
        return F.log_softmax(x, dim=1)
```

6. Optimization Configuration

To train the model, we utilize the **Adam Optimizer**, an adaptive learning rate optimization algorithm that is highly effective for deep learning tasks.

- **Learning Rate (lr=0.01):** Controls the step size during gradient descent.
- **Loss Function:** We will use Negative Log Likelihood Loss (nll_loss), which is appropriate when the model outputs log-probabilities.

```
model = GraphSAGEModel()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

7. Training Procedure

The model is trained over **200 epochs**. In every iteration, the model performs the following steps:

1. **Forward Pass:** Computes node embeddings and predictions based on current weights.
2. **Loss Calculation:** Measures the discrepancy between the predictions and the true labels (y).
3. **Backward Pass:** Calculates gradients via backpropagation.
4. **Parameter Update:** The optimizer updates the weights to minimize the loss.

```
for epoch in range(200):
   optimizer.zero_grad()       # Clear gradients
   out = model(x, edge_index)  # Forward pass
   loss = F.nll_loss(out, y)   # Calculate loss
   loss.backward()             # Backpropagation
   optimizer.step()            # Update weights
```

8. Evaluation and Results

After training is complete, we evaluate the model's performance by generating predictions on the nodes. We use the argmax function to select the class with the highest probability.

```
pred = out.argmax(dim=1)
print("Final Predictions:", pred)
```

Output Interpretation:
The output tensor([0, 0, 0, 1, 1, 1]) indicates:

- Nodes 0, 1, 2 are classified as **Class 0 (Benign)**.
- Nodes 3, 4, 5 are classified as **Class 1 (Malicious)**.